Техническая специализация Java

Иван Игоревич Овчинников

2024-05-12 (20:49)

Оглавление

1 Java Core **2**
1.1 Платформа: история и окружение . . . 2
1.1.1 Краткая история (причины возникновения) . . . 2
1.1.2 Базовый инструментарий, который понадобится (выбор IDE) . . . 3
1.1.3 Что нужно скачать, откуда (как выбрать вендора, версии) . . . 3
1.1.4 Из чего всё состоит (JDK, JRE, JVM и их друзья) . . . 5
1.1.5 Структура проекта (пакеты, классы, метод main, комментарии) . . . 8
1.1.6 Отложим мышки в сторону (CLI: сборка, пакеты, запуск) . . . 10
1.1.7 Документирование (Javadoc) . . . 11
1.1.8 Автоматизируй это (Makefile, Docker) . . . 12
1.2 Специализация: данные и функции . . . 22
1.2.1 Данные . . . 22
1.2.2 Примитивные типы данных . . . 23
1.2.3 Ссылочные типы данных, массивы . . . 33
1.2.4 Базовый функционал языка . . . 35
1.2.5 Функции . . . 38
1.3 Специализация: ООП . . . 47
1.3.1 Классы и объекты, поля и методы, статика . . . 47
1.3.2 Устройство памяти. Стек, куча и сборка мусора . . . 55
1.3.3 Конструкторы . . . 59
1.3.4 Инкапсуляция . . . 63
1.3.5 Наследование . . . 65
1.3.6 Полиморфизм . . . 73
1.4 Специализация: ООП и исключения . . . 84
1.4.1 Перечисления . . . 84
1.4.2 Внутренние и вложенные классы . . . 87
1.4.3 Исключения . . . 92
1.5 Специализация: тонкости работы . . . 110
1.5.1 Файловая система . . . 110
1.5.2 Файловая система и представление данных . . . 114
1.5.3 Потоки ввода-вывода, пакет `java.io` . . . 118
1.5.4 `java.nio` и `nio2` . . . 125
1.5.5 String . . . 127

2 Java Development Kit **138**
2.1 Инструментарий: GUI – графический интерфейс пользователя . . . 138
2.1.1 Почему именно Swing? . . . 139
2.1.2 JFrame: Главный класс окна . . . 139
2.1.3 Компоненты окна . . . 142

2.1.4 Многооконное приложение, взаимосвязи 147
2.1.5 Основная панель с игрой 151
2.2 Инструментарий: программные интерфейсы 168
2.2.1 Введение и результат 168
2.2.2 Подготовка проекта 169
2.2.3 Рисуемые объекты 175
2.2.4 Интерфейсы 178
2.2.5 Анонимные классы 187
2.2.6 Исключения в графических интерфейсах пользователя 192
2.3 Инструментарий: Обобщения 203
2.3.1 Понятие обобщения 203
2.3.2 Варианты обобщений 207
2.3.3 Ограниченные параметры типа 210
2.3.4 Выведение типов 214
2.3.5 Целевые типы 216
2.3.6 Вопросы для самопроверки 217
2.3.7 Подстановочный символ <?> (wildcard) 217
2.3.8 Стирание типа и загрязнение кучи 222
2.3.9 Ограничения обобщений (резюме) 223

Семинары **233**
А Семинар: компиляция и интерпретация кода 233
А.1 Инструментарий 233
А.2 Цели семинара 233
А.3 План-содержание 233
А.4 Подробности 234
Б Семинар: данные и функции 246
Б.1 Инструментарий 246
Б.2 Цели семинара 246
Б.3 План-содержание 246
Б.4 Подробности 247
В Семинар: классы и объекты 256
В.1 Инструментарий 256
В.2 Цели семинара 256
В.3 План-содержание 256
В.4 Подробности 257
Приложения 267
Г Семинар: обработка исключений 272
Г.1 Инструментарий 272
Г.2 Цели семинара 272
Г.3 План-содержание 272
Г.4 Подробности 273
Приложения 282
Д Семинар: тонкости работы 289
Д.1 Инструментарий 289
Д.2 Цели семинара 289
Д.3 План-содержание 289
Д.4 Подробности 290
Е Семинар: Простейшие интерфейсы пользователя 301
Е.1 Инструментарий 301
Е.2 Цели семинара 301

Е.3 План-содержание 301
Е.4 Подробности 302
Ж Семинар: Интерфейсы и API 318
Ж.1 Инструментарий 318
Ж.2 Цели семинара 318
Ж.3 План-содержание 318
Ж.4 Подробности 319
З Семинар: Обобщения 330
З.1 Инструментарий 330
З.2 Цели семинара 330
З.3 План-содержание 330
З.4 Подробности 331

1. Java Core

1.1. Платформа: история и окружение

В этом разделе

Краткая история (причины возникновения); инструментарий, выбор версии; CLI; структура проекта; документирование; некоторые интересные способы сборки проектов.

В этом разделе происходит первое знакомство с внутреннем устройством языка Java и фреймворком разработки приложений с его использованием. Рассматривается примитивный инструментарий и базовые возможности платформы для разработки приложений на языке Java. Разбирается структура проекта, а также происходит ознакомление с базовым инструментарием для разработки на Java.

- JDK
- JRE
- JVM
- JIT
- CLI
- Docker

1.1.1. Краткая история (причины возникновения)

- Язык создавали для разработки встраиваемых систем, сетевых приложений и прикладного ПО;
- Популярен из-за скорости исполнения и полного абстрагирования от исполнителя кода;
- Часто используется для программирования бэк-энда веб-приложений из-за изначальной нацеленности на сетевые приложения.

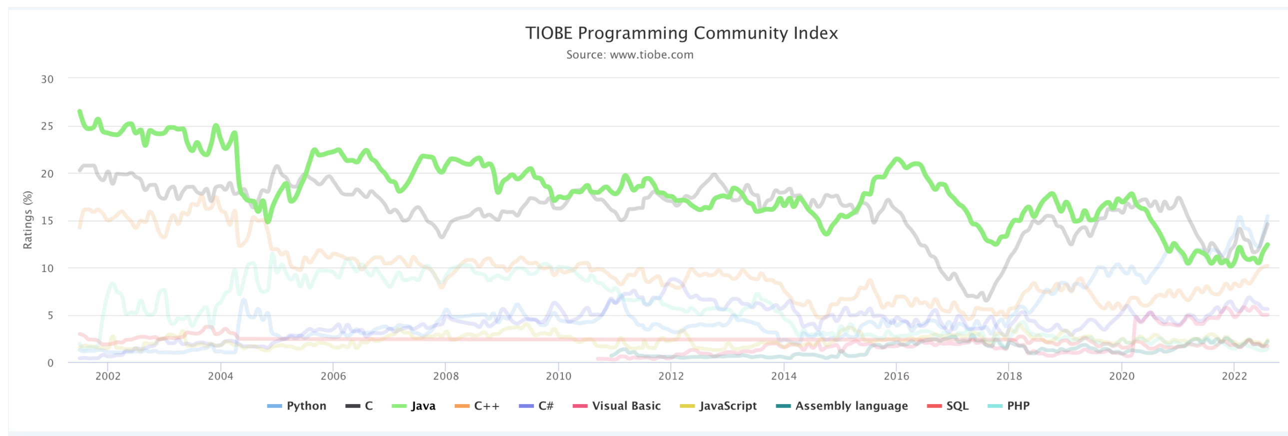


Рис. 1.1: График популярности языков программирования TIOBE

Задания для самопроверки

1. Как Вы думаете, почему язык программирования Java стал популярен в такие короткие сроки?
   - существовавшие на тот момент Pascal и C++ были слишком сложными;
   - Java быстрее C++;
   - однажды написанная на Java программа работает везде.

1.1.2. Базовый инструментарий, который понадобится (выбор IDE)

- NetBeans - хороший, добротный инструмент с лёгким ностальгическим оттенком;
- Eclipse - для поклонников Eclipse Foundation и швейцарских ножей с полусотней лезвий;
- IntelliJ IDEA - стандарт де-факто, используется на курсе и в большинстве современных компаний;
- Android Studio - если заниматься мобильной разработкой.

Задания для самопроверки

1. Как Вы думаете, почему среда разработки IntelliJ IDEA стала стандартом де-факто в коммерческой разработке приложений на Java?
   - NetBeans перестали поддерживать;
   - Eclipse слишком медленный и тяжеловесный;
   - IDEA оказалась самой дружелюбной к начинающему программисту;
   - Все варианты верны.

1.1.3. Что нужно скачать, откуда (как выбрать вендора, версии)

Для разработки понадобится среда разработки (IDE) и инструментарий разработчика (JDK). JDK выпускается несколькими поставщиками, большинство из них бесплатны и полнофункциональны, то есть поддерживают весь функционал языка и платформы.

В последнее время, с развитием контейнеризации приложений, часто устанавливают инструментарий в Docker-контейнер и ведут разработку прямо в контейнере, это позволяет не захламлять компьютер разработчика разными версиями инструментария и быстро разворачивать свои приложения в CI или на целевом сервере.

В общем случае, для разработки на любом языке программирования нужны так называемые SDK (Software Development Kit, англ. - инструментарий разработчика приложений или инструментарий для разработки приложений). Частный случай такого SDK - инструментарий разработчика на языке Java - Java Development Kit.

На курсе будет использоваться BellSoft Liberica JDK 11, но возможно использовать и других производителей, например, самую распространённую Oracle JDK. Производителя следует выбирать из требований по лицензированию, так, например, Oracle JDK можно использовать бесплатно только в личных целях, за коммерческую разработку с использованием этого инструментария придётся заплатить.

Для корректной работы самого инструментария и сторонних приложений, использующих инструментарий, проследите, пожалуйста, что установлены следующие переменные среды ОС:

- в системную `PATH` добавить путь до исполняемых файлов JDK, например, для UNIX-подобных систем: `PATH=$PATH:/usr/lib/jvm/jdk1.8.0_221/bin`
- `JAVA_HOME` путь до корня JDK, например, для UNIX-подобных систем: `JAVA_HOME=/usr/lib/jvm/jdk1.8.0_221/`
- `JRE_HOME` путь до файлов JRE из состава установленной JDK, например, для UNIX-подобных систем: `JRE_HOME=/usr/lib/jvm/jdk1.8.0_221/jre/`
- `J2SDKDIR` устаревшая переменная для JDK, используется некоторыми старыми приложениями, например, для UNIX-подобных систем: `J2SDKDIR=/usr/lib/jvm/jdk1.8.0_221/`
- `J2REDIR` устаревшая переменная для JRE, используется некоторыми старыми приложениями, например, для UNIX-подобных систем: `J2REDIR=/usr/lib/jvm/jdk1.8.0_221/jre/`

Также возможно использовать и другие версии, но не старше 1.8. Это обосновано тем, что основные разработки на данный момент только начинают обновлять инструментарий до более новых версий (часто 11 или 13) или вовсе переходят на другие JVM-языки, такие как Scala, Groovy или Kotlin.

Иногда для решения вопроса менеджмента версий прибегают к стороннему инструментарию, такому как SDKMan.

Для решения некоторых несложных задач курса мы будем писать простые приложения, не содержащие ООП, сложных взаимосвязей и проверок, в этом случае нам понадобится Jupyter notebook с установленным ядром (kernel) IJava.

Задания для самопроверки

1. Чем отличается SDK от JDK?
2. Какая версия языка (к сожалению) остаётся самой популярной в разработке на Java?
3. Какие ещё JVM языки существуют?

1.1.4. Из чего всё состоит (JDK, JRE, JVM и их друзья)

TL;DR:

- JDK = JRE + инструменты разработчика;
- JRE = JVM + библиотеки классов;
- JVM = Native API + механизм исполнения + управление памятью.

Как именно всё работает? Если коротко, то слой за слоем накладывая абстракции. Программы на любом языке программирования исполняются на компьютере, то есть, так или иначе, задействуют процессор, оперативную память и прочие аппаратурные компоненты. Эти аппаратурные компоненты предоставляют для доступа к себе низкоуровневые интерфейсы, которые задействует операционная система, предоставляя в свою очередь интерфейс чуть проще программам, взаимодействующим с ней. Этот интерфейс взаимодействия с ОС мы для простоты будем называть Native API.

С ОС взаимодействует JVM (Wikipedia: Список виртуальных машин Java), то есть, используя Native API, нам становится всё равно, какая именно ОС установлена на компьютере, главное уметь выполняться на JVM. Это открывает простор для создания целой группы языков, они носят общее бытовое название JVM-языки, к ним относят Scala, Groovy, Kotlin и другие. Внутри JVM осуществляется управление памятью, существует механизм исполнения программ, специальный JIT[1]-компилятор, генерирующий платформенно-зависимый код.

JVM для своей работы запрашивает у ОС некоторый сегмент оперативной памяти, в котором хранит данные программы. Это хранение происходит «слоями»:

1. Eden Space (heap) – в этой области выделяется память под все создаваемые из программы объекты. Большая часть объектов живёт недолго (итераторы, временные объекты, используемые внутри методов и т.п.), и удаляются при выполнении сборок мусора это области памяти, не перемещаются в другие области памяти. Когда данная область заполняется (т.е. количество выделенной памяти в этой области превышает некоторый заданный процент), сборщик мусора выполняет быструю (minor collection) сборку. По сравнению с полной сборкой, она занимает мало времени, и затрагивает только эту область памяти, а именно, очищает от устаревших объектов Eden Space и перемещает выжившие объекты в следующую область.
2. Survivor Space (heap) – сюда перемещаются объекты из предыдущей области после того, как они пережили хотя бы одну сборку мусора. Время от времени долгоживущие объекты из этой области перемещаются в Tenured Space.
3. Tenured (Old) Generation (heap) — Здесь скапливаются долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и прочие). Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.
4. Permanent Generation (non-heap) – Здесь хранится метаинформация, используемая JVM (используемые классы, методы и т.п.).
5. Code Cache (non-heap) — эта область используется JVM, когда включена JIT-компиляция, в ней кешируется скомпилированный платформенно-зависимый код.

JVM самостоятельно осуществляет сборку так называемого мусора, что значительно облегчает работу программиста по отслеживанию утечек памяти, но важно помнить, что в Java утечки памяти всё равно существуют, особенно при программировании многопоточных приложений.

[1] JIT, just-in-time - англ. вóвремя, прямо сейчас

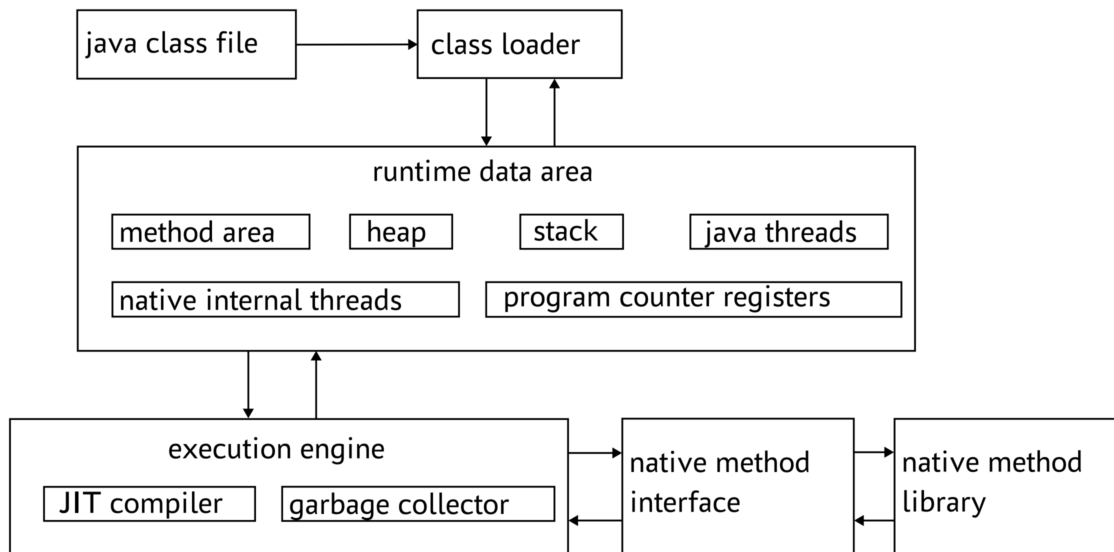


Рис. 1.2: Принцип работы JVM

На пользовательском уровне важно не только исполнять базовые инструкции программы, но чтобы эти базовые инструкции умели как-то взаимодействовать со внешним миром, в том числе другими программами, поэтому JVM интегрирована в JRE - Java Runtime Environment. JRE - это набор из классов и интерфейсов, реализующих

- возможности сетевого взаимодействия;
- рисование графики и графический пользовательский интерфейс;
- мультимедиа;
- математический аппарат;
- наследование и полиморфизм;
- рефлексию;
- ... многое другое.

Java Development Kit является изрядно дополненным специальными Java приложениями SDK. JDK дополняет JRE не только утилитами для компиляции, но и утилитами для создания документации, отладки, развёртывания приложений и многими другими. В таблице 1.1 на странице 7, приведена примерная структура и состав JDK и JRE, а также указаны их основные и наиболее часто используемые компоненты из состава Java Standard Edition. Помимо стандартной редакции существует и Enterprise Edition, содержащий компоненты для создания веб-приложений, но JEE активно вытесняется фреймворками Spring и Spring Boot.

Language							
tools + tools api	javac	java	javadoc	javap	jar	JPDA	
	JConsole	JavaVisualVM	JMC	JFR	Java DB	Int'l	JVM TI
	IDL	Troubleshoot	Security	RMI	Scripting	Web services	Deploy
deployment	Java Web			Applet/Java plug-in			
UI toolkit	Swing		Java 2D		AWT	Accessibility	
	Drag'n'Drop		Input Methods		Image I/O	Print Service	Sound
Integration libraries	IDL	JDBC	JNDI	RMI	RMI-IIOP	Scripting	
Other base libraries	Override Mechanism		Intl Support		Input/Output		JMX
	XML JAXP		Math		Networking		Beans
	Security		Serialization		Extension Mechanism		JNI
Java lang and util base libs	JAR	Lang and util	Ref Objects		Preference API		Reflection
	Zip	Management	Instrumentation		Stream API		Collections
	Logging	Regular Expressions	Concurrency Utilities		Datetime		Versioning
JVM	Java Hot Spot VM (JIT)						

Java Standard Edition
Java Runtime Environment
Java Development Kit

Таблица 1.1: Общее представление состава JDK

Задания для самопроверки

1. JVM и JRE - это одно и тоже?
2. Что входит в состав JDK, но не входят в состав JRE?
3. Утечки памяти
   – Невозможны, поскольку работает сборщик мусора;

- Возможны;
- Существуют только в C++ и других языках с открытым менеджментом памяти.

1.1.5. Структура проекта (пакеты, классы, метод main, комментарии)

Проекты могут быть любой сложности. Часто структуру проекта задаёт сборщик проекта, предписывая в каких папках будут храниться исходные коды, исполняемые файлы, ресурсы и документация. Без их использования необходимо задать структуру самостоятельно.

Простейший проект чаще всего состоит из одного файла исходного кода, который возможно скомпилировать и запустить как самостоятельный объект. Отличительная особенность в том, что чаще всего это один или несколько статических методов в одном классе.

Файл `Main.java` в этом случае может иметь следующий, минималистичный вид

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Скриптовый проект это достаточно новый тип проектов, он получил развитие благодаря растущей популярности Jupyter Notebook. Скриптовые проекты удобны, когда нужно отработать какую-то небольшую функциональность или пошагово пояснить работу какого-то алгоритма.

+ Code + Markdown | Run All | Clear Outputs of All Cells | Restart | Interrupt | Outline ... Java

0. Исходные данные

```
int[] arr = {1,0,1,1,0,0,0,1,1,1};
System.out.println(Arrays.toString(arr));
```

[1] ✓ 0.5s Java

```
[1, 0, 1, 1, 0, 0, 0, 1, 1, 1]
```

1. Очевидно

что если отнять от единицы единицу, результатом будет ноль, а если отнять от единицы ноль, то результатом останется единица

```
for (int i = 0; i < arr.length; ++i)
    arr[i] = 1 - arr[i];
System.out.println(Arrays.toString(arr));
```

[2] ✓ 0.1s Java

```
[0, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

2. Не так очевидно

Рис. 1.3: Пример простого Java проекта в Jupyter Notebook

Обычный проект состоит из пакетов, которые содержат классы, которые в свою очередь как-то связаны между собой и содержат код, который исполняется.

- Пакеты. Пакеты объединяют классы по смыслу. Классы, находящиеся в одном пакете доступны друг другу даже если находятся в разных проектах. У пакетов есть правила именования: обычно это обратное доменное имя (например, для gb.ru это будет ru.gb), название

проекта, и далее уже внутренняя структура. Пакеты именуют строчными латинскими буквами. Чтобы явно отнести класс к пакету, нужно прописать в классе название пакета после оператора `package`.

– Классы. Основная единица исходного кода программы. Одному файлу следует сопоставлять один класс. Название класса - это имя существительное в именительном падеже, написанное с заглавной буквы. Если требуется назвать класс в несколько слов, применяют UpperCamelCase.
– `public static void main(String[] args)`. Метод, который является точкой входа в программу. Должен находиться в публичном классе. При создании этого метода важно полностью повторить его сигнатуру и обязательно написать его с название со строчной буквы.
– Комментарии. Это часть кода, которую игнорирует компилятор при преобразовании исходного кода. Комментарии бывают:
  - `// comment` - до конца строки. Самый простой и самый часто используемый комментарий.
  - `/* comment */` - внутристрочный или многострочный. Никогда не используйте его внутри строк, несмотря на то, что это возможно.
  - `/** comment */` - комментарий-документация. Многострочный. Из него утилитой Javadoc создаётся веб-страница с комментарием.

Для примера был создан проект, содержащий два класса, находящихся в разных пакетах. Дерево проекта представлено на рис. 1.15, где папка с выходными (скомпилированными) бинарными файлами пуста, а файл README.md создан для лучшей демонстрации корня проекта.

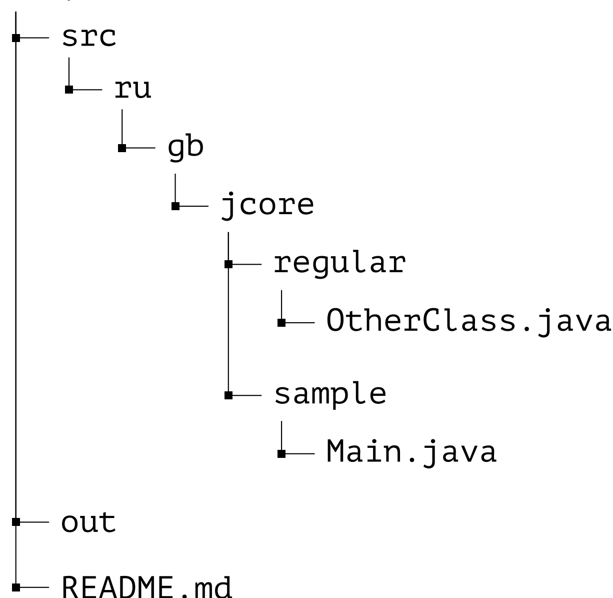


Рис. 1.4: Структура простого проекта

Содержимое файлов исходного кода представлено ниже.

```
1 package ru.gb.jcore.sample;
2
3 import ru.gb.jcore.regular.OtherClass;
4
5 public class Main {
6     public static void main(String[] args) {
```

```
        System.out.println("Hello, world!"); // greetings
        int result = OtherClass.sum(2, 2); // using a class from other package
        System.out.println(OtherClass.decorate(result));
    }
}
```

```
package ru.gb.jcore.regular;

public class OtherClass {
    public static int sum(int a, int b) {
        return a + b; // return without overflow check
    }

    public static String decorate(int a) {
        return String.format("Here is your number: %d.", a);
    }
}
```

Задания для самопроверки

1. Зачем складывать классы в пакеты?
2. Может ли существовать класс вне пакета?
3. Комментирование кода
   - Нужно только если пишется большая подключаемая библиотека;
   - Хорошая привычка;
   - Захламляет исходники.

1.1.6. Отложим мышки в сторону (CLI: сборка, пакеты, запуск)

Простейший проект возможно скомпилировать и запустить без использования тяжеловесных сред разработки, введя в командной строке ОС две команды:

- javac <Name.java> скомпилирует файл исходников и создаст в этой же папке файл с байт-кодом;
- java Name запустит скомпилированный класс (из файла с расширением `.class`).

```
ivan-igorevich@gb sources % ls
Main.java
ivan-igorevich@gb sources % javac Main.java
ivan-igorevich@gb sources % ls
Main.class  Main.java
ivan-igorevich@gb sources % java Main
Hello, world!
```

Скомпилированные классы всегда содержат одинаковые первые четыре байта, которые в шестнадцатиричном представлении формируют надпись «кофе, крошка».

```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789abcdef
00000000: cafe babe 0000 0037 001d 0a00 0600 0f09
00000010: 0010 0011 0800 120a 0013 0014 0700 1507
00000020: 0016 0100 063c 696e 6974 3e01 0003 2829
00000030: 5601 0004 436f 6465 0100 0f4c 696e 654e
00000040: 756d 6265 7254 6162 6c65 0100 046d 6169
00000050: 6e01 0016 285b 4c6a 6176 612f 6c61 6e67
00000060: 2f53 7472 696e 673b 2956 0100 0a53 6f75
```

Для компиляции более сложных проектов, необходимо указать компилятору, откуда забирать файлы исходников и куда складывать готовые файлы классов, а интерпретатору, откуда забирать файлы скомпилированных классов. Для этого существуют следующие ключи:

- javac:
  - `-d` выходная папка (директория) назначения;
  - `-sourcepath` папка с исходниками проекта;
- java:
  - `-classpath` папка с классами проекта;

Классы проекта компилируются в выходную папку с сохранением иерархии пакетов.

```
ivan-igorevich@gb Sample % javac -sourcepath ./src -d out src/ru/gb/jcore/sample/Main.java
ivan-igorevich@gb Sample % java -classpath ./out ru.gb.jcore.sample.Main
Hello, world!
Here is your number: 4.
```

Задания для самопроверки

1. Что такое javac?
2. Кофе, крошка?
3. Где находится класс в папке назначения работы компилятора?
   - В подпапках, повторяющих структуру пакетов в исходниках
   - В корне плоским списком;
   - Зависит от ключей компиляции.

1.1.7. Документирование (Javadoc)

Документирование конкретных методов и классов всегда ложится на плечи программиста, потому что никто не знает программу и алгоритмы в ней лучше, чем программист. Утилита Javadoc избавляет программиста от необходимости осваивать инструменты создания веб-страниц и записывать туда свою документацию. Достаточно писать хорошо отформатированные комментарии, а остальное Javadoc возьмёт на себя.

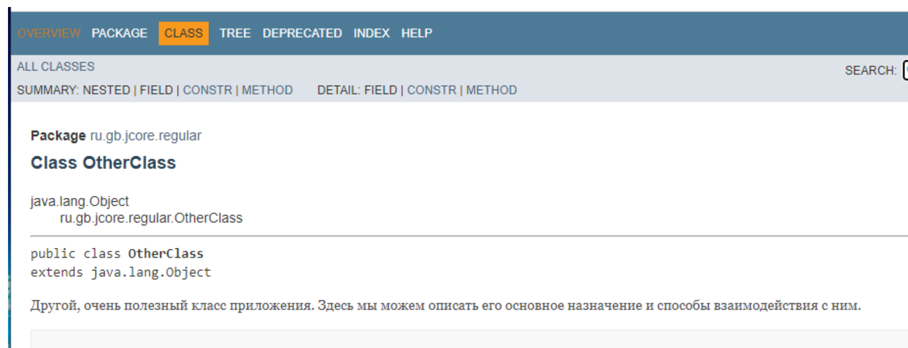


Рис. 1.5: Часть страницы автосгенерированной документации

Чтобы просто создать документацию надо вызвать утилиту javadoc с набором ключей.

- `ru` пакет, для которого нужно создать документацию;
- `-d` папка (или директория) назначения;
- `-sourcepath` папка с исходниками проекта;
- `-cp` путь до скомпилированных классов;
- `-subpackages` нужно ли заглядывать в пакеты-с-пакетами;

Часто необходимо указать, в какой кодировке записан файл исходных кодов, и в какой кодировке должна быть выполнена документация (например, файлы исходников на языке Java всегда сохраняются в кодировке UTF-8, а основная кодировка для ОС Windows - cp1251)

- `-locale ru_RU` язык документации (для правильной расстановки переносов и разделяющих знаков);
- `-encoding` кодировка исходных текстов программы;
- `-docencoding` кодировка конечной сгенерированной документации.

Чаще всего в комментариях используются следующие ключевые слова:

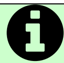
- `@param` описание входящих параметров
- `@throws` выбрасываемые исключения
- `@return` описание возвращаемого значения
- `@see` где ещё можно почитать по теме
- `@since` с какой версии продукта доступен метод
- `{@code "public"}` вставка кода в описание

Задания для самопроверки

1. Javadoc находится в JDK или JRE?
2. Что делает утилита Javadoc?
   - Создаёт комментарии в коде;
   - Создаёт программную документацию;
   - Создаёт веб-страницу с документацией из комментариев.

1.1.8. Автоматизируй это (Makefile, Docker)

В подразделе 1.1.6 мы проговорили о сборке проектов вручную. Компилировать проект таким образом — занятие весьма утомительное, особенно когда исходных файлов становится много, в проект включаются библиотеки и прочее.



`Makefile` — это набор инструкций для программы `make` (классическая, это GNU Automake), которая помогает собирать программный проект в одну команду. Если запустить `make` то программа попытается найти файл с именем по-умолчанию `Makefile` в текущем каталоге и выполнить инструкции из него.

Make, не привносит ничего принципиально нового в процесс компиляции, а только лишь автоматизируют его. В простейшем случае, в `Makefile` достаточно описать так называемую цель, `target`, и что нужно сделать для достижения этой цели. Цель, собираемая по-умолчанию называется `all`, так, для простейшей компиляции нам нужно написать:

```
all:
    javac -sourcepath .src/ -d out src/ru/gb/jcore/sample/Main.java
```

Внимание поклонникам войны за пробелы против табов в тексте программы: в Makefile для отступов при описании таргетов нельзя использовать пробелы. Только табы. Иначе make обнаруживает ошибку синтаксиса.

По сути, это всё. Но возможно сделать более гибко настраиваемый файл, чтобы не нужно было запоминать, как называются те или иные папки и файлы. В Makefile можно записывать переменные, например:

- SRCDIR := src
- OUTDIR := out

И далее вызывать их (то есть подставлять их значения в нужное место текста) следующим образом:

```
javac -sourcepath .${SRCDIR}/ -d ${OUTDIR}
```

Чтобы вызвать утилиту для сборки цели по-умолчанию, достаточно в папке, содержащей `Makefile` в терминале написать `make`. Чтобы воспользоваться другими написанными таргетами нужно после имени утилиты написать через пробел название таргета

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут на любой системе, поддерживающей соответствующую технологию.

Docker также не привносит ничего технологически нового, но даёт возможность не устанавливать JDK и не думать о переключении между версиями, достаточно взять контейнер с нужной версией инструментария и запустить приложение в нём.

Образы и контейнеры создаются с помощью специального файла, имеющего название `Dockerfile`. Первой строкой `Dockerfile` мы обязательно должны указать, какой виртуальный образ будет для нас основой. Здесь можно использовать как образы ОС, так и образы SDK.

```
  FROM bellsoft/liberica-openjdk-alpine:11.0.16.1-1
```

При создании образа необходимо скопировать все файлы из папки `src` проекта внутрь образа, в папку `src`.

```
  COPY ./src ./src
```

Потом, также при создании образа, надо будет создать внутри папку `out` простой терминальной командой, чтобы компилятору было куда складывать готовые классы.

```
RUN mkdir ./out
```

Последнее, что будет сделано при создании образа - запущена компиляция.

```
RUN javac -sourcepath ./src -d out ./src/ru/gb/dj/Main.java
```

Последняя команда в `Dockerfile` говорит, что нужно сделать, когда контейнер создаётся из образа и заускается.

```
CMD java -classpath ./out ru.gb.dj.Main
```

Docker-образ и, как следствие, Docker-контейнеры возможно настроить таким образом, чтобы скомпилированные файлы находились не в контейнере, а складывались обратно на компьютер пользователя через общие папки.

Часто команды разработчиков эмулируют таким образом реальный продакшн сервер, используя в качестве исходного образа не JDK, а образ целевой ОС, вручную устанавливают на ней JDK, запуская далее своё приложение.

Домашнее задание

- Создать проект из трёх классов (основной с точкой входа и два класса в другом пакете), которые вместе должны составлять одну программу, позволяющую производить четыре основных математических действия и осуществлять форматированный вывод результатов пользователю;
- Скомпилировать проект, а также создать для этого проекта стандартную веб-страницу с документацией ко всем пакетам;
- Создать Makefile с задачами сборки, очистки и создания документации на весь проект.
- *Создать два Docker-образа. Один должен компилировать Java-проект обратно в папку на компьютере подьзователя, а второй забирать скомпилированные классы и исполнять их.

Термины, определения и сокращения

`finally` часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.

`throws` ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.

`throw` оператор, активирующий (выбрасывающий) объект исключения.

`try...catch` двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.

BIOS (англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.

CI (англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CLI (англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».

cp1251 набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.

Docker программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

1.2. Специализация: данные и функции

В предыдущем разделе

- Краткая история (причины возникновения);
- инструментарий, выбор версии;
- CLI;
- структура проекта;
- документирование;
- некоторые интересные способы сборки проектов.

В этом разделе

Будет рассмотрен базовый функционал языка, то есть основная встроенная функциональность, такая как математические операторы, условия, циклы, бинарные операторы. Далее способы хранения и представления данных в Java, и в конце способы манипуляции данными, то есть функции (в терминах языка называющиеся методами).

- Метод;
- Типизация;
- Переполнение;
- Инициализация;
- Идентификатор;
- Typecasting;
- Массив;

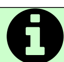
1.2.1. Данные

Понятие типов

Хранение данных в Java осуществляется привычным для программиста образом: в переменных и константах.

Относительно типизации языки программирования бывают типизированными и нетипизированными (бестиповыми). Нетипизированные языки не представляют большого интереса в современном программировании.

Отсутствие типизации в основном присуще чрезвычайно старым и низкоуровневым языкам программирования, например, Forth и некоторым ассемблерам. Все данные в таких языках считаются цепочками бит произвольной длины и не делятся на типы. Работа с ними часто труднее, при этом часто безтиповые языки работают быстрее типизированных, но описывать с их помощью большие проекты со сложными взаимосвязями довольно утомительно.

 Java является языком со **строгой** (также можно встретить термин «**сильной**») **явной статической** типизацией.

- Статическая - у каждой переменной должен быть тип, и этот тип изменить нельзя. Этому свойству противопоставляется динамическая типизация;

- Явная - при создании переменной ей обязательно необходимо присвоить какой-то тип, явно написав это в коде. В более поздних версиях языка (с 9й) стало возможным инициализировать переменные типа `var`, обозначающий нужный тип тогда, когда его возможно однозначно вывести из значения справа. Бывают языки с неявной типизацией, например, Python;
- Строгая(сильная) - невозможно смешивать разнотипные данные. С другой стороны, существует JavaScript, в котором запись `2 + true` выдаст результат 3.

Антипаттерн «магические числа»

Почти во всех примерах, которые используются для обучения, можно увидеть так называемый антипаттерн - плохой стиль для написания кода. Числа, которые находятся справа от оператора присваивания используются в коде без пояснений. Такой антипаттерн называется «магическое число». Магическое, потому что непонятно, что это за число, почему это число именно такое и что будет, если это число изменить.

Так лучше не делать. Заранее нужно сказать, что рекомендуется помещать все числа в коде в именованные константы, которые хранятся в начале файла. Плюсом такого подхода является возможность легко корректировать значения переменных в достаточно больших проектах.

Например, в вашем коде несколько тысяч строк, а какое-то число, скажем, возраст совершеннолетия, число 18, использовалось несколько десятков раз. При использовании приложения в стране, где совершеннолетием считается 21 год вы должны будете перечитывать весь код в поисках магических «18» и исправить их на «21». В этом вопросе будет также важно не запутаться, действительно ли это 18, которые означают совершеннолетие, а не количество карманов в жилетке Анатолия Вассермана[2].

В случае с константой изменить число нужно в одном месте.

1.2.2. Примитивные типы данных

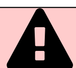
Все данные в Java делятся на две основные категории: примитивные и ссылочные. Таблица 1.2 демонстрирует все восемь примитивных типов языка и их размерности. Чтобы отправить на хранение какие-то данные используется оператор присваивания. Присваивание в программировании - это не тоже самое, что математическое равенство, демонстрирующее тождественность, а полноценная операция.

Все присваивания всегда происходят справа налево, то есть сначала вычисляется правая часть, а потом результат вычислений присваивается левой. Исключений нет, именно поэтому в левой части не может быть никаких вычислений.

Шесть из восьми типов имеет диапазон значений, а значит основное их отличие в объёме занимаемой памяти. У `double` и `float` тоже есть диапазоны, но они заключаются в точности представления дробной части. Диапазоны означают, что если попытаться положить в переменную меньшего типа большее значение, произойдёт «переполнение переменной».

Переполнение целочисленных переменных

Чем именно чревато переполнение переменной легче показать на примере (по ссылке - расследование крушения ракеты из-за переполнения переменной)



Переполнение переменных не распознаётся компилятором.

[2] мы то знаем, что их 26

Тип	Пояснение	Диапазон
byte	Самый маленький из адресуемых типов, 8 бит, знаковый	[−128, +127]
short	Тип короткого целого числа, 16 бит, знаковый	[−32 768, +32 767]
char	Целочисленный тип для хранения символов в кодировке UTF-8, 16 бит, беззнаковый	[0, +65 535]
int	Основной тип целого числа, 32 бита, знаковый	[−2 147 483 648, +2 147 483 647]
long	Тип длинного целого числа, 64 бита, знаковый	[−9 223 372 036 854 775 808, +9 223 372 036 854 775 807]
float	Тип вещественного числа с плавающей запятой (одинарной точности, 32 бита)	
double	Тип вещественного числа с плавающей запятой (двойной точности, 64 бита)	
boolean	Логический тип данных	true, false

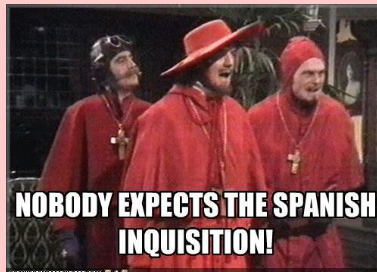
Таблица 1.2: Основные типы данных в языке Java

Если создать переменную типа byte, диапазон которого от $[-128, +127]$, и присвоить этой переменной значение 200 произойдёт переполнение, как если попытаться влить пакет молока в напёрсток.

Переполнение переменной - это ситуация, в которой происходит попытка положить большее значение в переменную меньшего типа.

Важным вопросом при переполнении остаётся следующий: какое в переполненной переменной останется значение? Максимальное, 127? $200 - 127 = 73$? Какой-то мусор? Каждый язык, а зачастую и разные компиляторы одного языка ведут себя в этом вопросе по разному.

В современном мире гигагерцев и терабайтов почти никто не пользуется маленькими типами, но именно из-за этого ошибки переполнения переменных становятся опаснее испанской инквизиции.



Задание для самопроверки

1. Возможно ли объявить в Java целочисленную переменную и присвоить ей дробное значение?

2. Магическое число - это:
   (a) числовая константа без пояснений;
   (b) число, помогающее в вычислениях;
   (c) числовая константа, присваиваемая при объявлении переменной.
3. Переполнение переменной - это:
   (a) слишком длинное название переменной;
   (b) слишком большое значение переменной;
   (c) расширение переменной вследствие записи большого значения.

Бинарное (битовое) представление данных

После разговора о переполнении, нельзя не сказать о том, что именно переполняется. Далее будут представлены сведения которые касаются не только языка Java но и любого другого языка программирования. Эти сведения помогут разобраться в деталях того как хранится значение переменной в программе и как, в целом, происходит работа компьютерной техники.

Все современные компьютеры, так или иначе работают от электричества и являются примитивными по своей сути устройствами, которые понимают только два состояния: есть напряжение в электрической цепи или нет. Эти два состояния принято записывать в виде 1 и 0, соответственно.

Все данные в любой программе - это единицы и нули. Данные в программе на Java не исключение, удобнее всего это явление рассматривать на примере примитивных данных. Поскольку в компьютере можно оперировать только двумя значениями то естественным образом используется двоичная система счисления.

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
00	00000	00	0x00
01	00001	01	0x01
02	00010	02	0x02
03	00011	03	0x03
04	00100	04	0x04
05	00101	05	0x05
06	00110	06	0x06
07	00111	07	0x07
08	01000	10	0x08
09	01001	11	0x09
10	01010	12	0x0a
11	01011	13	0x0b
12	01100	14	0x0c
13	01101	15	0x0d
14	01110	16	0x0e
15	01111	17	0x0f
16	10000	20	0x10

Таблица 1.3: Представления чисел

Двоичная система счисления это система счисления с основанием два. Существуют и другие системы счисления, например, восьмеричная, но сейчас она отходит на второй план полностью уступая своё место шестнадцатеричной системе счисления. Каждая цифра в десятичной

записи числа называется разрядом, аналогично в двоичной записи чисел каждая цифра тоже называется разрядом, но для компьютерной техники этот разряд называется битом.

Одна единица или ноль - это один **бит** передаваемой или хранимой информации.

Биты принято собирать в группы по восемь штук, по восемь разрядов, эти группы называются **байт**. В языке Java возможно оперировать минимальной единицей информации, такой как байт для этого есть соответствующий тип. Диапазон байта, согласно таблицы $[-128, +127]$, то есть байт информации может в себе содержать ровно 256 значений. Само число 127 в двоичной записи это семиразрядное число, все разряды которого единицы (то есть байт выглядит как `01111111`). Последний, восьмой, самый старший бит, определяет знак числа[3]. Достаточно знать формулу расчёта записи отрицательных значений:

1. в прямой записи поменять все нули на единицы и единицы на нули;
2. поставить старший бит в единицу.

Так возможно получить на единицу меньшее отрицательное число, то есть преобразовав 0 получим -1, 1 будет -2, 2 станет -3 и так далее.

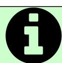
Числа бо́льших разрядностей могут хранить бо́льшие значения, теперь преобразование диапазонов из десятичной системы счисления в двоичную покажет что `byte` это один байт, `short` это два байта, то есть 16 бит, `int` это 4 байта то есть 32 бита, а `long` это 8 байт или 64 бита хранения информации.

Задания для самопроверки

1. Возможно ли число 3000000000 (3 миллиарда) записать в двоичном представлении?
2. Как вы думаете, почему шестнадцатеричная система счисления вытеснила восьмеричную?

Целочисленные типы

Целочисленных типов четыре, и они занимают 1, 2, 4 и 8 байт.



Технически, целочисленых типов пять, но `char` устроен чуть сложнее других, поэтому не рассматривается в этом разделе.

Значения в целочисленных типах могут быть только целые, никак и никогда невозможно присвоить им дробных значений. Про эти типы следует помнить следующее:

- `int` - это самый часто используемый тип. Если сомневаетесь, какой целочисленный тип использовать, используйте `int`;
- все целые числа, которые пишутся в коде - это `int`, даже если вы пытаетесь их присвоить переменной другого типа.

Как `int` преобразуется в меньше типы? Если написать цифрами справа число, которое может поместиться в переменную меньшего типа слева, то статический анализатор кода его пропустит, а компилятор преобразует в меньший тип автоматически (строка 9 на рис. 1.6).

[3] Здесь можно начать долгий и скучный разговор о схемотехнике и хранении отрицательных чисел с применением техники дополнительного кода.

```
9        byte b0 = 100;
10       byte b1 = 200;
11                        Required type:   byte
12                        Provided:        int
13                        Cast to 'byte'  ⌥⇧↩     More actions...  ⌥↩
14
```

Рис. 1.6: Присваивание валидных и переполняющих значений

Как видно, к маленькому `byte` успешно присваивается `int`. Если же написать число которое больше типа слева и, соответственно, поместиться не может, среда разработки выдает предупреждение компилятора, что ожидался `byte`, а передан `int` (строка 10 рис 1.6).

Часто нужно записать в виде числа какое-то значение большее чем может принимать `int`, и явно присвоить начальное значение переменной типа `long`.

```
9        byte b0 = 100;
10       byte b1 = 200;
11       long l0 = 5_000_000_000;
12                   Integer number too large
13
```

Рис. 1.7: Попытка инициализации переменной типа `long`

В примере на рис. 1.7 показана попытка присвоить значение 5000000000 переменной типа `long`. Из текста ошибки ясно, что невозможно положить такое большое значение в переменную типа `int`, а это значит, что справа `int`. Почему большой `int` без проблем присваивается к маленькому байту?

```
9        byte b0 = 100;
10       byte b1 = 200;
11       long l0 = 5_000_000_000;
12       long l1 = 5_000_000_000L;
13       float f0 = 0.123;
14       float f1 = 0.123f;
```

Рис. 1.8: Решение проблемы переполнения числовых констант

На рис. 1.8 продемонстрировано, что аналогичная ситуация возникает с типами `float` и `double`. Все дробные числа, написанные в коде - это `double`, поэтому положить их во `float` без дополнительных усилий невозможно. В этих случаях к написанному справа числу нужно

добавить явное указание на его тип. Для `long` пишем `L`, а для `float` - `f`. Чаще всего `L` пишут заглавную, чтобы подчеркнуть, что тип больше, а `f` пишут маленькую, чтобы подчеркнуть, что мы уменьшаем тип. Но регистр в этом конкретном случае значения не имеет, можно писать и так и так.

Числа с плавающей запятой (точкой)

Как видно из таблицы 1.2, два из восьми типов не имеют диапазонов значений. Это связано с тем, что диапазоны значений флоута и дабла заключаются не в величине возможных хранимых чисел, а в точности этих чисел после запятой.

> Числа с плавающей запятой в англоязычной литературе называются числа с плавающей точкой (от англ. floating point). Такое различие связано с тем, что в русскоязычной литературе принято отделять дробную часть числа запятой, а в европейской и американской - точкой.

Хранение чисел с плавающей запятой[4] работает по стандарту IEEE 754 (1985 г). Для работы с числами с плавающей запятой на аппаратурном уровне к обычному процессору добавляют математический сопроцессор (FPU, floating point unit).

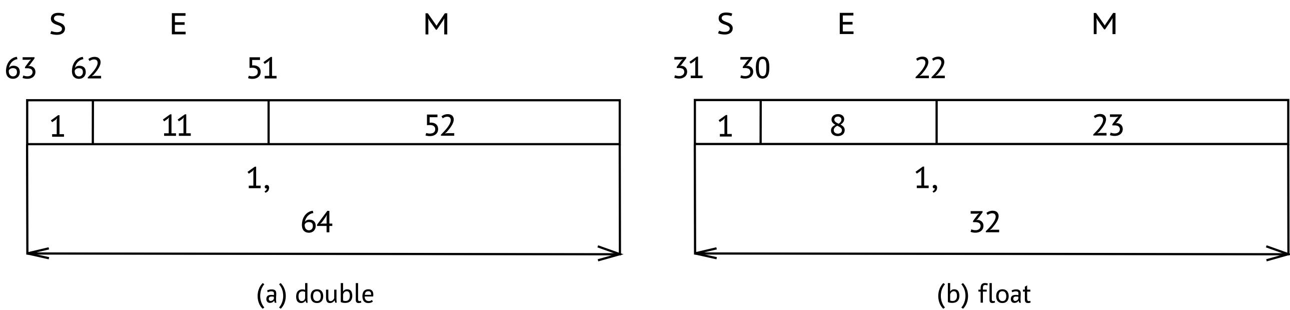


Рис. 1.9: Типы с плавающей запятой

Рисунок 1.9 демонстрирует, как распределяются биты в числах с плавающей запятой разных разрядностей, где S - Sign (знак), E - Exponent (8(11) разрядов поля порядка, экспонента), M - Mantissa (23(52) бита мантиссы, дробная часть числа).

Если попытаться уложить весь стандарт в два предложения, то получится примерно следующее: получить число в соответствующих разрядностях возможно по формулам:

$$F_{32} = (-1)^S \times 2^{E-127} \times (1 + \frac{M}{2^{23}})$$

$$F_{64} = (-1)^S \times 2^{E-1023} \times (1 + \frac{M}{2^{52}})$$

[4] хорошо и подробно, но на C, в посте на Хабре.

Например: $+0,5 = 2^{-1}$ поэтому, число будет записано как

`0_01111110_00000000000000000000000`, то есть знак $= 0$, мантисса $= 0$, порядок = $127 - 1 = 126$, чтобы получить следующие результаты вычислений:

-1^0 положительный знак, умножить на порядок
$2^{126-127=-1} = 0,5$ и умножить на мантиссу
$1 + 0$. То есть, $-1^0 \times 2^{-1} \times (1 + 0) = 0,5$.
Отсюда становится очевидно, что чем сложнее мантисса и чем меньше порядок, тем более точные и интересные числа мы можем получить.

Возьмём для примера число $-0,15625$, чтобы понять как его записывать, откинем знак, это будет единица в разряде, отвечающем за знак, и посчитаем мантиссу с порядком. Представим число как положительное и будем от него последовательно отнимать числа, являющиеся отрицательными степенями двойки, чтобы получить максимально близкое к нулю значение.

$$2^1 = 2$$
$$2^0 = 1.0$$
$$2^{-1} = 0.5$$
$$2^{-2} = 0.25$$
$$2^{-3} = 0.125$$
$$2^{-4} = 0.0625$$
$$2^{-5} = 0.03125$$
$$2^{-6} = 0.015625$$
$$2^{-7} = 0.0078125$$
$$2^{-8} = 0.00390625$$

Очевидно, что -1 и -2 степени отнять не получится, поскольку мы явно уходим за границу нуля, а вот -3 прекрасно отнимается, значит порядок будет $127 - 3 = 124$, осталось понять, что получится в мантиссе.

Видим, что оставшееся после первого вычитания ($0,15625 - 0,125$) число - это 2^{-5}. Значит в мантиссе пишем `01` и остальные нули, то есть слева направо указываем, какие степени после -3 будут нужны. -4 не нужна, а -5 нужна.

Получится, что

$$(-1)^1 \times 2^{(124-127)} \times (1 + \frac{2097152}{2^{23}}) = 1,15652$$

или, тождественно,

$$(-1)^1 \times 1,01e - 3 =$$
$$1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} =$$
$$1 \times 0,125 + 0 \times 0,0625 + 1 \times 0,03125 =$$
$$0,125 + 0,03125 = 0,15625$$

Так число с плавающей запятой возможно посчитать двумя способами: по приведённой формуле, или последовательно складывая разряды мантиссы умноженные на двойку в степени порядка, уменьшая порядок на каждом шагу.

К особенностям работы чисел с плавающей запятой можно отнести:

– возможен как положительный, так и отрицательный ноль (в целых числах ноль всегда положительный);

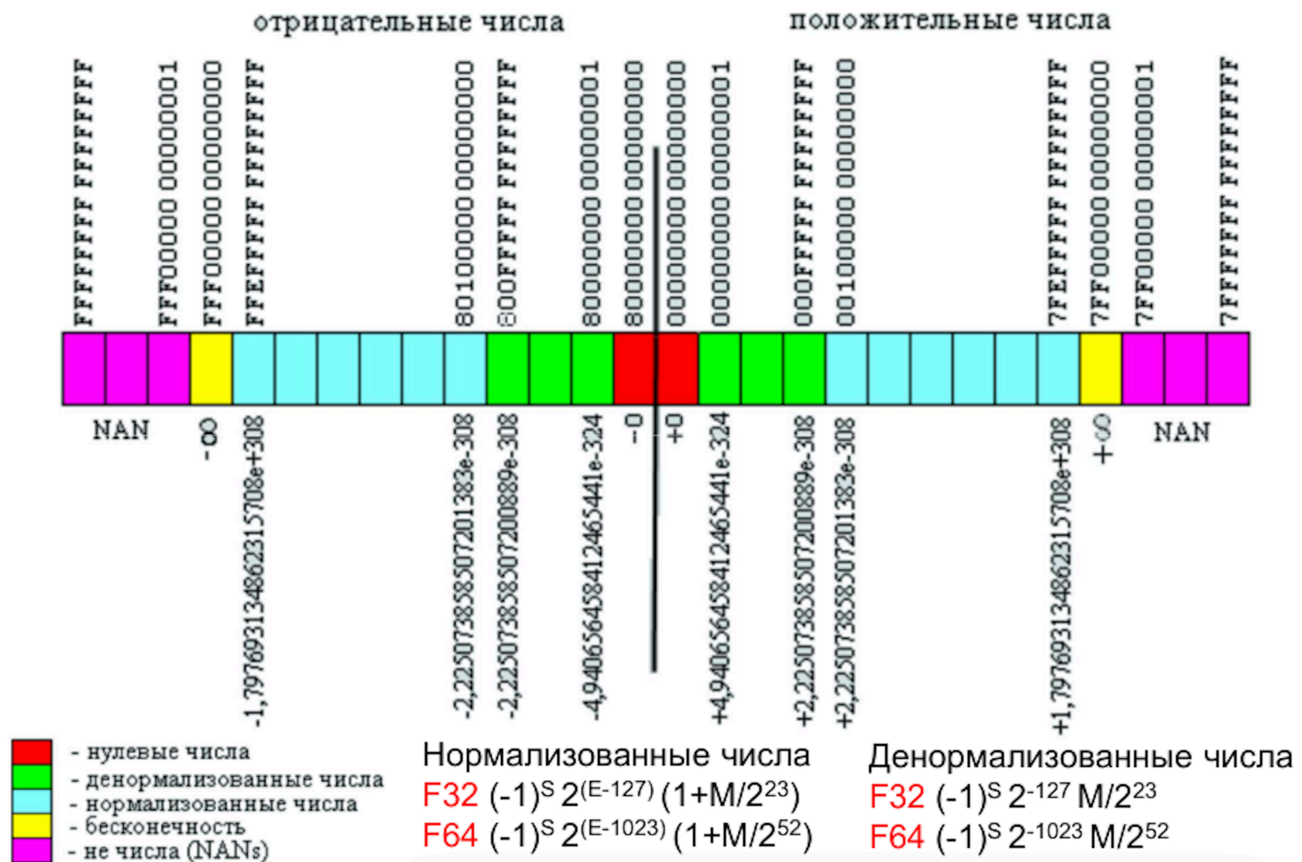


Рис. 1.10: Особенности работы с числами с плавающей запятой

- есть огромная зона, отмеченная на рисунке 1.10, которая являет собой непредставимые числа, слишком большие для хранения внутри такой переменной или настолько маленькие, что мнимая единица в мантиссе отсутствует;
- в таком числе можно хранить значения положительной и отрицательной бесконечности;
- при работе с такими числами появляется понятие не-числа, при этом важно помнить, что `NaN != NaN`.

Задания для самопроверки

1. Сколько байт данных занимает самый большой целочисленный тип?
2. Почему нельзя напрямую сравнивать целочисленные данные и числа с плавающей запятой, даже если там точно лежит число без дробной части?
3. Внутри переполненной переменной остаётся значение:
   (a) переданное - максимальное для типа;
   (b) максимальное для типа;
   (c) не определено.

Символы и булевы

Шесть из восьми примитивных типов могут иметь как положительные, так и отрицательные значения, они называются **«знаковые»** типы. В таблице есть два типа, у которых есть диапазон но нет отрицательных значений, это `boolean` и `char`

Булев тип хранит значение `true` или `false`. На собеседованиях иногда спрашивают, сколько места занимает `boolean`. В Java объём хранения не определён и зависит от конкретной JVM, обычно считают, что это один байт.

dec	hex	val	dec	hex	val	dec	hex	val	dec	hex	val
000	0x00	(nul)	032	0x20	□	064	0x40	@	096	0x60	‘
001	0x01	(soh)	033	0x21	!	065	0x41	A	097	0x61	a
002	0x02	(stx)	034	0x22	”	066	0x42	B	098	0x62	b
003	0x03	(etx)	035	0x23	#	067	0x43	C	099	0x63	c
004	0x04	(eot)	036	0x24	$	068	0x44	D	100	0x64	d
005	0x05	(enq)	037	0x25	%	069	0x45	E	101	0x65	e
006	0x06	(ack)	038	0x26	&	070	0x46	F	102	0x66	f
007	0x07	(bel)	039	0x27	'	071	0x47	G	103	0x67	g
008	0x08	(bs)	040	0x28	(	072	0x48	H	104	0x68	h
009	0x09	(tab)	041	0x29	)	073	0x49	I	105	0x69	i
010	0x0A	(lf)	042	0x2A	*	074	0x4A	J	106	0x6A	j
011	0x0B	(vt)	043	0x2B	+	075	0x4B	K	107	0x6B	k
012	0x0C	(np)	044	0x2C	’	076	0x4C	L	108	0x6C	l
013	0x0D	(cr)	045	0x2D	-	077	0x4D	M	109	0x6D	m
014	0x0E	(so)	046	0x2E	.	078	0x4E	N	110	0x6E	n
015	0x0F	(si)	047	0x2F	/	079	0x4F	O	111	0x6F	o
016	0x10	(dle)	048	0x30	0	080	0x50	P	112	0x70	p
017	0x11	(dc1)	049	0x31	1	081	0x51	Q	113	0x71	q
018	0x12	(dc2)	050	0x32	2	082	0x52	R	114	0x72	r
019	0x13	(dc3)	051	0x33	3	083	0x53	S	115	0x73	s
020	0x14	(dc4)	052	0x34	4	084	0x54	T	116	0x74	t
021	0x15	(nak)	053	0x35	5	085	0x55	U	117	0x75	u
022	0x16	(syn)	054	0x36	6	086	0x56	V	118	0x76	v
023	0x17	(etb)	055	0x37	7	087	0x57	W	119	0x77	w
024	0x18	(can)	056	0x38	8	088	0x58	X	120	0x78	x
025	0x19	(em)	057	0x39	9	089	0x59	Y	121	0x79	y
026	0x1A	(eof)	058	0x3A	:	090	0x5A	Z	122	0x7A	z
027	0x1B	(esc)	059	0x3B	;	091	0x5B	[	123	0x7B	{
028	0x1C	(fs)	060	0x3C	<	092	0x5C	\	124	0x7C	|
029	0x1D	(gs)	061	0x3D	=	093	0x5D	]	125	0x7D	}
030	0x1E	(rs)	062	0x3E	>	094	0x5E	^	126	0x7E	~
031	0x1F	(us)	063	0x3F	?	095	0x5F	_	127	0x7F	\DEL

Таблица 1.4: Фрагмент UTF-8 (ASCII) таблицы

Тип `char` единственный беззнаковый целочисленный тип в языке, то есть его старший разряд хранит полезное значение, а не признак положительности. Тип целочисленный но по умолчанию среда исполнения интерпретирует его как символ по таблице utf-8 (см фрагмент в таблице 1.4). В языке Java есть разница между одинарными и двойными кавычками. В одинарных кавычках всегда записывается символ, который на самом деле является целочисленным значением, а в двойных кавычках всегда записывается строка, которая фактически является экземпляром класса `String`. Поскольку типизация строгая, то невозможно записать в `char` строки, а в строки числа.

В Java есть три основных понятия, связанных с данными переменными и использованием значений: объявление, присваивание, инициализация.

Для того чтобы *объявить* переменную, нужно написать её тип и название, также часто вместо названия можно встретить термин идентификатор.
Далее в любой момент можно *присвоить* этой переменной значение, то есть необходимо написать идентификатор использовать оператор присваивания и справа написать значение, которое вы хотите присвоить данной переменной, поставить в конце строки точку с запятой.
Также существует понятие *инициализации* - это когда объединяются на одной строке объявление и присваивание.

Преобразование типов

Java - это язык со строгой статической типизацией, но преобразование типов в ней всё равно есть. Простыми словами, преобразование типов - это когда компилятор видит, что типы переменных по разные стороны присваивания разные, начинает разрешать это противоречие. Преобразование типов бывает явное и неявное.

В разговоре или в сообществах можно услышать или прочитать термины тайпкастинг, кастинг, каст, кастануть, и другие производные от английского typecasting.

Неявное преобразование типов происходит, когда присваиваются числа переменным меньшей размерности, чем `int`. Число справа это `int`, а значит 32 разряда, а слева, например, `byte`, и в нём всего 8 разрядов, но ни среда ни компилятор не поругались, потому что значение в большом `int` не превысило 8 разрядов маленького `byte`. Итак неявное преобразование типов происходит в случаях, когда, «всё и так понятно». В случае, если неявное преобразование невозможно, статический анализатор кода выдаёт ошибку, что ожидался один тип, а был дан другой.

Явное преобразование типов происходит, когда мы явно пишем в коде, что некоторое значение должно иметь определённый тип. Этот вариант приведения типов тоже был рассмотрен, когда к числам дописывались типовые квалификаторы `L` и `f`. Но чаще всего случается, что происходит присваивание переменным не тех значений, которые были написаны в тексте программы, а те, которые получились в результате каких-то вычислений.

```
9     int i0 = 100;
10    byte b0 = i0;
11
```

Рис. 1.11: Ошибка приведения типов

На рис. 1.11 приведён простейший пример, в котором очевидно, что внутри переменной `i0` содержится значение, не превышающее одного байта хранения, а значит возможно явно

сообщить компилятору, что значение точно поместится в `byte`. *Явно преобразовать типы*. Для этого нужно в правой части оператора присваивания перед идентификатором переменной в скобках добавить название типа, к которому необходимо преобразовать значение этой переменной.

```
9       int i0 = 100;
10      byte b0 = (byte) i0;
11
12
13
14
```

Рис. 1.12: Верное приведение типов

Константность

Constare - (лат. стоять твёрдо). Константность это свойство неизменяемости. В Java ключевое слово `const` не реализовано, хоть и входит в список ключевых, зарезервированных. Константы создаются при помощи ключевого слова `final`. Ключевое слово файнал возможно применять не только с примитивами, но и со ссылочными типами, методами, классами.

Константа - это переменная или идентификатор с конечным значением.

Задания для самопроверки

1. Какая таблица перекодировки используется для представления символов?
2. Каких действий требует от программиста явное преобразование типов?
3. какое значение будет содержаться в переменной a после выполнения строки int a = 10.0f/3.0f;

1.2.3. Ссылочные типы данных, массивы

Ссылочные типы данных - это все типы данных, кроме восьми перечисленных примитивных. Самым простым из ссылочных типов является массив. Фактически массив выведен на уровень языка и не имеет специального ключевого слова.

Ссылочные типы отличаются от примитивных местом хранения информации. В примитивах данные хранятся там, где существует переменная и где написан её идентификатор, а по идентификатору ссылочного типа хранится не значение, а ссылка. Ссылку можно представить как ярлык на рабочем столе, то есть очевидно, что непосредственная информация хранится не там, где написан идентификатор. Такое явное разделение идентификатора переменной и данных важно помнить и понимать при работе с ООП.

Массив - это единая, сплошная область данных, в связи с чем в массивах возможно осуществление доступа по индексу

Самый младший индекс любого массива - ноль, поскольку **индекс** - это значение смещения по элементам относительно начального адреса массива. То есть, для получения самого первого элемента нужно сместиться на ноль шагов. Очевидно, что самый последний элемент в массиве из десяти значений, будет храниться по девятому индексу.

Массивы возможно создавать несколькими способами (листинг 1.1). В общем виде объявление - это тип, квадратные скобки как обозначение того, что это будет массив из переменных этого типа, идентификатор (строка 1). Инициализировать массив можно либо ссылкой на другой массив (строка 2), пустым массивом (строка 3) или заранее заданными значениями, записанными через запятую в фигурных скобках (строка 4). Присвоить в процессе работы идентификатору возможно только значение ссылки из другого идентификатора или новый пустой массив.

Листинг 1.1: Объявление массива

```
1 int[] array0;
2 int[] array1 = array0;
3 int[] array2 = new int[5];
4 int[] array3 = {5, 4, 3, 2, 1};
5
6 array2 = {1, 2, 3, 4, 5}; //
```

Никак и никогда нельзя присвоить идентификатору целый готовый массив в процессе работы, нельзя стандартными средствами переприсвоить ряд значений части массива (так называемые слайсы или срезы).

Массивы бывают как одномерные, так и многомерные. Многомерный массив - это всегда массив из массивов меньшего размера: двумерный массив - это массив одномерных, трёхмерный - массив двумерных и так далее. Правила инициализации у них не отличаются. Преобразовать тип массива нельзя никогда, но можно преобразовать тип каждого отдельного элемента при чтении. Это связано с тем, что под массивы сразу выделяется непрерывная область памяти, а со сменой типа всех значений массива эту область нужно будет или значительно расширять или значительно сужать.

Ключевое слово `final` работает только с идентификатором массива, то есть не запрещает изменять значения его элементов.

Если логика программы предполагает создание нижних измерений массива в процессе работы программы, то при инициализации массива верхнего уровня не следует указывать размерности нижних уровней. Это связано с тем, что при инициализации, Java сразу выделяет память под все измерения, а присваивание нижним измерениям новых ссылок на создаваемые в процессе работы массивы, будет пересоздавать области памяти, получается небольшая утечка памяти.

Прочитать из массива значение возможно обратившись к ячейке массива по индексу. Записать в массив значение возможно обратившись к ячейке массива по индексу, и применив оператор присваивания.

```
1 int i = array[0];
2 array[1] = 10;
```

В каждом объекте массива есть специальное поле (рис. 1.13), которое обозначает длину данного массива. Поле находится в классе `__Array__` и является публичной константой.

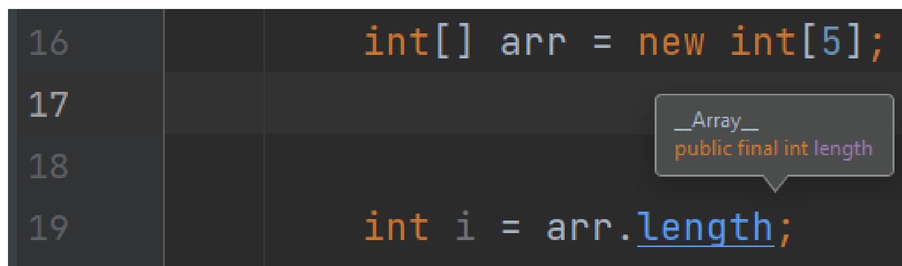


Рис. 1.13: Константа с длиной массива

Задания для самопроверки

1. Почему индексация массива начинается с нуля?
2. Какой индекс будет последним в массиве из 100 элементов?
3. Сколько будет создано одномерных массивов при инициализации массива 3x3?

1.2.4. Базовый функционал языка

Математические операторы

Математические операторы работают как и предполагается - складывают, вычитают, делят, умножают, делают это по приоритетам известным нам с пятого класса, а если приоритет одинаков - слева направо. Специального оператора возведения в степень как в пайтоне нет. Единственное, что следует помнить, что оператор присваивания продолжает быть оператором присваивания, а не является математическим равенством, а значит сначала посчитается всё, что слева, а потом результат попробует присвоиться переменной справа. Припоминаем что там за дела с целочисленным делением и отбрасыванием дробной части.

Условия

Условия представлены в языке привычными `if`, `else if`, `else`, «если», «иначе если», «в противном случае», которые являются единым оператором выбора, то есть если исполнение программы пошло по одной из веток, то в другую ветку условия программа точно не зайдёт. Каждая ветвь условного оператора - это отдельный кодовый блок со своим окружением и локальными переменными.

Существует альтернатива оператору `else if` - использование оператора `switch`, который позволяет осуществлять множественный выбор между числовыми значениями. У оператора есть ряд особенностей:

- это оператор, состоящий из одного кодового блока, то есть сегменты кода находятся в одной области видимости. Если не использовать оператор `break`, есть риск «проваливаться» в следующие кейсы;
- нельзя создать диапазон значений;
- достаточно сложно создавать локальные переменные с одинаковым названием для каждого кейса.

Циклы

Циклы представлены основными конструкциями:

- `while () {}`

- `do {} while();`
- `for (;;) {}`

Цикл - это набор повторяющихся до наступления условия действий. `while` - самый простой, чаще всего используется, когда нужно описать бесконечный цикл. `do-while` единственный цикл с постусловием, то есть сначала выполняется тело, а затем принимается решение о необходимости зацикливания, используется для ожидания ответов на запрос и возможного повторения запроса по условию. `for` - классический счётный цикл, его почему-то программисты любят больше всего.

Существует также активно пропагандируемый цикл - `foreach`, работает не совсем очевидным образом, для понимания его работы необходимо ознакомиться с ООП и понятием итератора.

Бинарные арифметические операторы

В современных реалиях мегамощных компьютеров вряд ли кто-то задумывается об оптимизации скорости выполнения программы или экономии занимаемой памяти. Но всё меняется, когда программист впервые принимает сложное решение: запрограммировать микроконтроллер или другой «интернет вещей». Там в вашем распоряжении жалкие пара сотен килобайт памяти, если очень повезёт, в которые нужно не только как-то вложить текст программы и исполняемый бинарный код, но и какие-то промежуточные, пользовательские и другие данные, буферы обмена и обработки. Другая ситуация, в которой нужно начинать «думать о занимаемом пространстве» это разработка протоколов передачи данных, чтобы протокол был быстрый, не передавал по сети большие объёмы данных и быстро преобразовывался. На помощь приходит натуральная для информатики система счисления, двоичная.

Манипуляции двоичными данными представлены в Джава следующими операторами:

- & битовое и;
- | битовое или;
- ~ битовое не;
- ^ исключающее или;
- << сдвиг влево;
- >> сдвиг вправо.

Литеральные «и», «или», «не» уже знакомы по условным операторам. Литеральные операции применяются ко всему числовому литералу целиком, а не к каждому отдельному биту. Их особенность заключается в том, как язык программирования интерпретирует числа.

В Java в литеральных операциях может участвовать только тип `boolean`, а C++ воспринимает любой ненулевой целочисленный литерал как истину, а нулевой, соответственно, как ложь.

Логика формирования значения при этом остаётся такой же, как и при битовых операциях.

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

A	NOT
0	1
1	0

Таблица 1.5: Таблицы истинности битовых операторов

Когда говорят о битовых операциях волей-неволей появляется необходимость поговорить о таблицах истинности. В таблице 1.5 вы видите таблицы истинности для арифметических битовых операций. Битовые операции отличаются тем, что для неподготовленного взгляда они производят почти магические действия, потому что манипулируют двоичным представлением числа.

```
 00000100   4      00000100   4      00000100   4
&00000111   7     |00000111   7     ^00000111   7     ~00000100   4
 00000100   4      00000111   7      00000011   3      11111011  -5
```

Рис. 1.14: Бинарная арифметика

Число	Бинарное	Сдвиг	Число	Бинарное	Сдвиг
2	000000010	2 << 0	128	010000000	128 >> 0
4	000000100	2 << 1	64	001000000	128 >> 1
8	000001000	2 << 2	32	000100000	128 >> 2
16	000010000	2 << 3	16	000010000	128 >> 3
32	000100000	2 << 4	8	000001000	128 >> 4
64	001000000	2 << 5	4	000000100	128 >> 5
128	010000000	2 << 6	2	000000010	128 >> 6

Таблица 1.6: Битовые сдвиги

С битовыми сдвигами работать гораздо интереснее и выгоднее. Они производят арифметический сдвиг значения слева на количество разрядов, указанное справа, в таблице 1.6 представлены числа, в битовом представлении это одна единственная единица, находящаяся в разных разрядах числа. Это демонстрация сдвига на один разряд влево, и, как следствие, умножение на два. Обратная ситуация со сдвигом вправо, он является целочисленным делением.

- X && Y - литеральная;
- X || Y - литеральная;
- !X - литеральная;
- N << K - $N * 2^K$;
- N >> K - $N/2^K$;
- x & y - битовая. 1 если оба x = 1 и y = 1;
- x | y - битовая. 1 если хотя бы один из x = 1 или y = 1;
- ~x - битовая. 1 если x = 0;
- x ^ y - битовая. 1 если x отличается от y.

Задания для самопроверки

1. Почему нежелательно использовать оператор `switch` если нужно проверить диапазон значений?
2. Возможно ли записать бесконечный цикл с помощью оператора `for`?
3. `2 + 2 * 2 == 2 << 2 >> 1`?

1.2.5. Функции

Функция - это исполняемый блок кода. Функция, принадлежащая классу называется **методом**.

```
void int method(int param1, int param2) {
    //function body
}

public static void main (String[] args) {
    method(arg1, arg2);
}
```

При объявлении функции в круглых скобках указываются параметры, а при вызове - аргументы.

У функций есть правила именования: функция - это переходный глагол совершенного вида в настоящем времени (вернуть, посчитать, установить, создать), часто снабжаемый дополнением, субъектом действия. Методы в Java пишутся lowerCamelCase. Важно, в каком порядке записаны параметры метода, от этого будет зависеть порядок передачи в неё аргументов. Методы обособлены и их параметры локальны, то есть не видны другим функциям.

Нельзя писать функции внутри других функций.

Все аргументы передаются копированием, не важно, копирование это числовой константы, числового значения переменной или хранимой в переменной ссылке на массив. Сам объект в метод не копируется, а копируется только его ссылка.

Возвращаемые из методов значения появляются в том месте, где метод был вызван. Если будет вызвано несколько методов, то весь контекст исполнения первого метода сохраняется, кладётся (на стек) в стопку уже вызванных методов и процессор идёт выполнять только что вызванный второй метод. По завершении вызванного второго метода, мы снимаем со стека лежащий там контекст первого метода, кладём в него вернувшееся из второго метода значение, если оно есть, и продолжаем исполнять первый метод.

Вызов метода - это, по смыслу, тоже самое, что подставить в код сразу его возвращаемое значение.

Сигнатура метода - это имя метода и его параметры. В сигнатуру метода не входит возвращаемое значение. Нельзя написать два метода с одинаковой сигнатурой.

Перегрузка методов - это механизм языка, позволяющий написать методы с одинаковыми названиями и разными оставшимися частями сигнатуры, чтобы получить единообразие при вызове семантически схожих методов с разнотипными данными.

Практическое задание

1. Написать метод «Шифр Цезаря», с булевым параметром зашифрования и расшифрования и числовым ключом;
2. Написать метод, принимающий на вход массив чисел и параметр n. Метод должен осуществить циклический (последний элемент при сдвиге становится первым) сдвиг всех элементов массива на n позиций;
3. Написать метод, которому можно передать в качестве аргумента массив, состоящий строго из единиц и нулей (целые числа типа `int`). Метод должен заменить единицы в массиве

на нули, а нули на единицы и не содержать ветвлений. Написать как можно больше вариантов метода.

Термины, определения и сокращения

`finally` часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.

`throws` ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.

`throw` оператор, активирующий (выбрасывающий) объект исключения.

`try...catch` двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.

BIOS (англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.

CI (англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CLI (англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».

cp1251 набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.

Docker программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT (англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.

GPL GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.

IDE (от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.

JDK (от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.

JIT (англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.

JRE (от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.

JVM (от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.

NTFS (аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

1.3. Специализация: ООП

В предыдущем разделе

Будет рассмотрен базовый функционал языка, то есть основная встроенная функциональность, такая как математические операторы, условия, циклы, бинарные операторы. Далее способы хранения и представления данных в Java, и в конце способы манипуляции данными, то есть функции (в терминах языка называющиеся методами).

В этом разделе

Разберём такие основополагающих в Java вещи, как классы и объекты, а также с тем, как применять на практике основные принципы ООП: наследование, полиморфизм и инкапсуляцию. Дополнительно рассмотрим устройство памяти в джава.

- Класс;
- Объект;
- Статика;
- Стек;
- Куча;
- Сборщик мусора;
- Конструктор;
- Инкапсуляция;
- Наследование;
- Полиморфизм ;

1.3.1. Классы и объекты, поля и методы, статика

Классы

Что такое класс? С точки зрения ООП, **класс** определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java.

Наиболее важная особенность класса состоит в том, что он определяет новый тип данных, которым можно воспользоваться для создания объектов этого типа

То есть класс — это шаблон (чертёж), по которому создаются объекты (экземпляры класса). Для определения формы и сущности класса указываются данные, которые он должен содержать, а также код, воздействующий на эти данные. Создаем мы свои классы, когда у нас не хватает уже созданных.

Например, если мы хотим работать в нашем приложении с документами, то необходимо для начала объяснить приложению, что такое документ, описать его в виде класса (чертежа) `Document`. Указать, какие у него должны быть свойства: название, содержание, количество страниц, информация о том, кем он подписан и т.п. В этом же классе мы обычно описываем, что можно делать с документами: печатать в консоль, подписывать, изменять содержание, название и т.д. Результатом такого описания и будет класс `Document`. Однако, это по-прежнему всего лишь чертеж хранимых данных (состояний) и способы взаимодействия с этими данными.

Если нам нужны конкретные документы, а нам они обязательно нужны, то необходимо создавать **объекты**: документ №1, документ №2, документ №3. Все эти документы будут иметь одну и ту же структуру (описанные нами название, содержание, ...), с ними можно выполнять одни и те же описанные нами действия (печатать, подписать, ...), но наполнение будет разным, например, в первом документе содержится приказ о назначении работника на должность, во втором, о выдаче премии отделу разработки и т.д.

Начнём с малого, напишем свой первый класс. Представим, что необходимо работать в приложении с котами. Java ничего не знает о том, что такое коты, поэтому необходимо создать новый класс (тип данных), и объяснить что такое кот. Создадим новый файл, для простоты в том же пакете, что и главный класс программы.

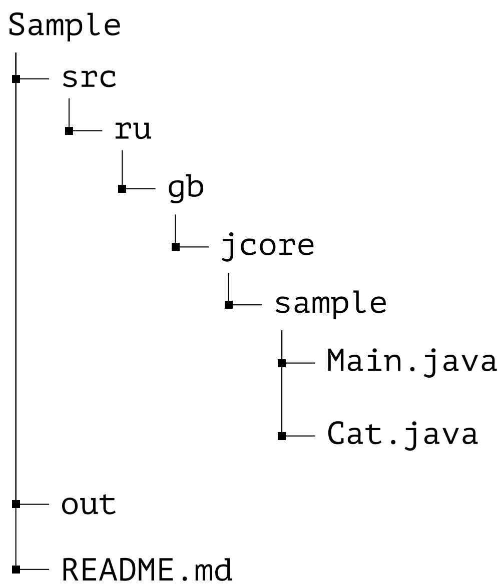


Рис. 1.15: Структура проекта

Поля класса

Начнем описывать в классе `Cat` так называемый API кота. Как изестно, имя класса должно совпадать с именем файла, в котором он объявлен, т.е. класс `Cat` должен находиться в файле `Cat.java`. Пусть у котов есть три свойства: `name` (кличка), `color` (цвет) и `age` (возраст); совокупность этих свойств называется состоянием, и коты пока ничего не умеют делать. Класс `Cat` будет иметь вид, представленный в листинге 1.2. Свойства класса, записанные таким образом, в виде переменных, называются **полями**.

Листинг 1.2: Структура кота в программе

```
package ru.gb.jcore;

public class Cat {
    String name;
    String color;
    int age;
}
```

Для новичка важно не запутаться, класс кота мы описали в отдельном файле, а создавать объекты и совершать манипуляции следует в основном классе программы, не может же кот назначить имя сам себе.

Мы рассказали программе, что такое коты, теперь если мы хотим создать в нашем приложении конкретного кота, следует воспользоваться оператором `new Cat();` в основном классе программы. Более подробно разберём, что происходит в этой строке, чуть позже, пока же нам достаточно знать, что мы создали объект типа `Cat` (экземпляр класса `Cat`), и запомнить эту конструкцию. Для того чтобы с ним (экземпляром) работать, можем положить его в переменную, которой дать идентификатор `cat1`. При создании объекта полям присваиваются значения по умолчанию (нули для числовых переменных и `false` для булевых).

```
Cat cat0; // cat0 = null;
cat0 = new Cat();
Cat cat1 = new Cat();
```

В листинге выше можно увидеть все три операции (объявление, присваивание и инициализацию) и становится понятно, как можно создавать объекты. Также известно, что в переменной не лежит сам объект, а только ссылка на него. Объект `cat1` создан по чертежу `Cat`, это значит, что у него есть поля `name`, `color`, `age`, с которыми можно работать: получать или изменять их значения.

Для доступа к полям объекта используется оператор точка, который связывает имя объекта с именем поля. Например, чтобы присвоить полю `color` объекта `cat1` значение «Белый», нужно выполнить код `cat1.color = "Белый";`

Операция «точка» служит для доступа к полям и методам объекта по его имени. Мы уже использовали оператор «точка» для доступа к полю с длиной массива, например. Рассмотрим пример консольного приложения, работающего с объектами класса `Cat`. Создадим двух котов, один будет белым Барсиком 4х лет, второй чёрным Мурзиком шести лет, и просто выведем информацию о них в терминал.

```java
package ru.gb.jcore;

public class Main {
   public static void main(String[] args) {
      Cat cat1 = new Cat();
      Cat cat2 = new Cat();

      cat1.name = "Barsik";
      cat1.color = "White";
      cat1.age = 4;

      cat2.name = "Murzik";
      cat2.color = "Black";
      cat2.age = 6;

      System.out.println("Cat1 named: " + cat1.name +
            " is " + cat1.color +
            " has age: " + cat1.age);
```

```java
        System.out.println("Cat2 named: " + cat2.name +
                " is " + cat2.color +
                " has age: " + cat2.age);
    }
}
```

в результате работы программы в консоли появятся следующие строки:

```
Cat1 named: Barsik is White has age: 4
Cat2 named: Murzik is Black has age: 6
```

Вначале мы создали два объекта типа Cat: cat1 и cat2, соответственно, они имеют одинаковый набор полей name, color, age. Почему? Потому что они принадлежат одному классу, созданы по одному шаблону. Объекты всегда «знают», какого они класса. Однако каждому из них в эти поля записаны разные значения. Как видно из результата печати в консоли, изменение значения полей одного объекта, никак не влияет на значения полей другого объекта. Данные объектов cat1 и cat2 изолированы друг от друга. А значит мы делаем вывод о том, поля хранятся в классе, а значения полей хранятся в объектах. Логическая структура, демонстрирующая отношения объектов и классов, в том числе в части хранения полей и их значений показана на рис. 1.16.

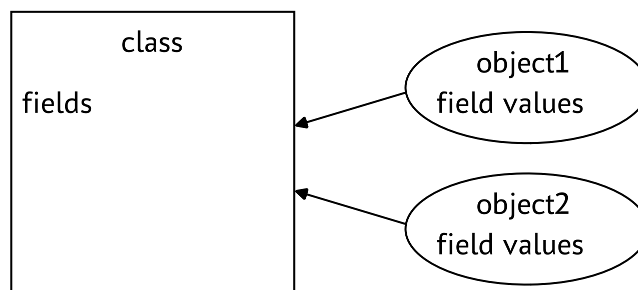


Рис. 1.16: Логическая структура отношения класс-объект

Объекты

Разобравшись с тем, как создавать новые типы данных (классы) и мельком посмотрев, как создаются объекты, нужно подробнее разобрать, как создавать объекты, и что при этом происходит. Создание объекта как любого ссылочного типа данных проходит в два этапа. Как и в случае с уже известными нам массивами.

- Сначала создается переменная, имеющая интересующий нас тип, в неё возможно записать ссылку на объект;
- затем необходимо выделить память под объект;
- создать и положить объект в выделенную часть памяти;
- и сохранить ссылку на этот объект в памяти - в нашу переменную.

Для непосредственного создания объекта применяется оператор new, который динамически резервирует память под объект и возвращает ссылку на него, в общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором new.

```java
Cat cat1; // cat1 = null;
cat1 = new Cat();
Cat cat2 = new Cat();
```

В первой строке кода переменная cat1 объявляется как ссылка на объект типа Cat и пока ещё не ссылается на конкретный объект (первоначально значение переменной cat1 равно null). В следующей строке выделяется память для объекта типа Cat, и в переменную cat1 сохраняется ссылка на него. После выполнения второй строки кода переменную cat1 можно использовать так, как если бы она была объектом типа Cat. Обычно новый объект создается в одну строку, то есть инициализируется.

Оператор new

[квалификаторы] ИмяКласса имяПеременной = **new** ИмяКласса();

Оператор new динамически выделяет память для нового объекта, общая форма применения этого оператора имеет вид как на врезке выше, но на самом деле справа - не имя класса, конструкция ИмяКласса() в правой части выполняет вызов конструктора данного класса, который подготавливает вновь создаваемый объект к работе.

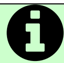
Именно от количества применений оператора new будет зависеть, сколько именно объектов будет создано в программе.

```
1  Cat cat1 = new Cat();
2  Cat cat2 = cat1;
3  
4  cat1.name = "Barsik";
5  cat1.color = "White";
6  cat1.age = 4;
7  
8  cat2.name = "Murzik";
9  cat2.color = "Black";
10 cat2.age = 6;
11 
12 System.out.println("Cat1 named: " + cat1.name +
13       " is " + cat1.color +
14       " has age: " + cat1.age);
15 System.out.println("Cat2 named: " + cat2.name +
16       " is " + cat2.color +
17       " has age: " + cat2.age);
```

На первый взгляд может показаться, что переменной cat2 присваивается ссылка на копию объекта cat1, т.е. переменные cat1 и cat2 будут ссылаться на разные объекты в памяти. Но это не так. На самом деле cat1 и cat2 будут ссылаться на один и тот же объект. Присваивание переменной cat1 значения переменной cat2 не привело к выделению области памяти или копированию объекта, лишь к тому, что переменная cat2 содержит ссылку на тот же объект, что и переменная cat1. Это явление дополнительно подчёркивает ссылочную природу данных в языке Java.

Таким образом, любые изменения, внесённые в объект по ссылке cat2, окажут влияние на объект, на который ссылается переменная cat1, поскольку *это один и тот же объект в памяти*. Поэтому результатом выполнения кода, где мы как будто бы указали возраст второго кота, равный шести годам, станут строки, показывающие, что по обеим ссылкам оказался кот возраста шесть лет с именем Мурзика.

```
Cat1 named: Murzik is Black has age: 6
Cat2 named: Murzik is Black has age: 6
```



Множественные ссылки на один и тот же объект в памяти довольно легко себе представить как ярлыки для запуска одной и той же программы на рабочем столе и в меню быстрого запуска. Или если на один и тот же шкафчик в раздевалке наклеить два номера - сам шкафчик можно будет найти по двум ссылкам на него.

Важно всегда перепроверять, какие объекты созданы, а какие имеют множественные ссылки.

Методы

Ранее было сказано о том, что в языке Java любая программа состоит из классов и функций, которые могут описываться только внутри них. Именно поэтому все функции в языке Java являются методами. А метод - это функция, являющаяся частью некоторого класса, которая может выполнять операции над данными этого класса.

Метод - это функция, принадлежащая классу

Метод для своей работы может использовать поля объекта и/или класса, в котором определен, напрямую, без необходимости передавать их во входных параметрах. Это похоже на использование глобальных переменных в функциях, но в отличие от глобальных переменных, метод может получать прямой доступ только к членам класса. Самые простые методы работают с данными объектов. Методы чаще всего формируют API классов, то есть способ взаимодействия с классами, интерфейс. Место методов во взаимодействии классов и объектов показано на рис. 1.17.

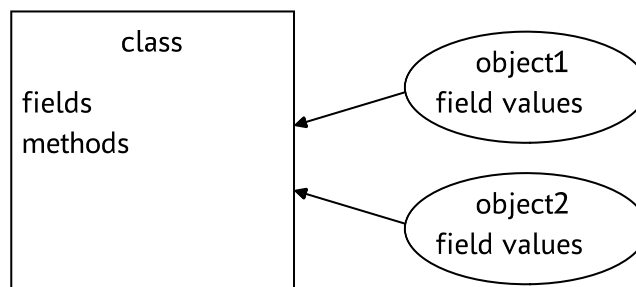


Рис. 1.17: Логическая структура отношения класс-объект

Вернёмся к примеру с котиками. Широко известно, что котики умеют урчать, мяукать и смешно прыгать. В целях демонстрации в описании этих действий просто будем делать разные выводы в консоль, хотя возможно и научить котика в программе выбирать минимальное значение из массива, но это было бы, как минимум, неожиданно. Итак опишем метод например подать голос и прыгать.

```
1 public class Cat {
2    String name;
3    String color;
4    int age;
5
6    void voice() {
7       System.out.println(name + " meows");
8    }
```

```
    void jump() {
        if (this.age < 5) System.out.println(name + " jumps");
    }
}
```

Обращение к методам выглядит очень похожим на стандартный способом, через точку, как к полям. Теперь когда появляется необходимость позвать котика, он скажет: «мяу, я имя котика», а если в программе пришло время котику прыгнуть, он решит, прилично ли это – прыгать в его возрасте.

```
package ru.gb.jcore;

public class Main {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
        Cat cat2 = new Cat();

        cat1.name = "Barsik";
        cat1.color = "White";
        cat1.age = 4;

        cat2.name = "Murzik";
        cat2.color = "Black";
        cat2.age = 6;

        cat1.voice();
        cat2.voice();
        cat1.jump();
        cat2.jump();
    }
}
```

```
Barsik meows
Murzik meows
Barsik jumps
```

Как видно, Барсик замечательно прыгает, а Мурзик от прыжков воздержался, хотя попрыгать программа попросила их обоих.

Ключевое слово `static`

В завершение базовой информации о классах и объектах, остановимся на специальном модификаторе `static`, делающем переменную или метод «независимыми» от объекта.

`static` – модификатор, применяемый к полю, блоку, методу или внутреннему классу, он указывает на привязку субъекта к текущему классу.

Для использования таких полей и методов, соответственно, объект создавать не нужно. В Java большинство членов служебных классов являются статическими. Возможно использовать это ключевое слово в четырех контекстах:

– статические методы;
– статические переменные;
– статические вложенные классы;
– статические блоки.

В этом разделе рассмотрим подробнее только первые два пункта, третий опишем чуть позже, а четвёртый потребует от нас знаний, выходящих не только за этот урок, но и за десяток следующих.

Статические методы также называются методами класса, потому что статический метод принадлежит классу, а не его объекту. Нестатические называются методами объекта. Статические методы можно вызывать напрямую через имя класса, не обращаясь к объекту и вообще объект не создавая. Что это и зачем нужно? Например, умение кота мяукать можно вывести в статическое поле, потому что, например, весной можно открыть окно, не увидеть ни одного экземпляра котов, но зато услышать их, и точно знать, что мяукают не дома и не машины, а именно коты.

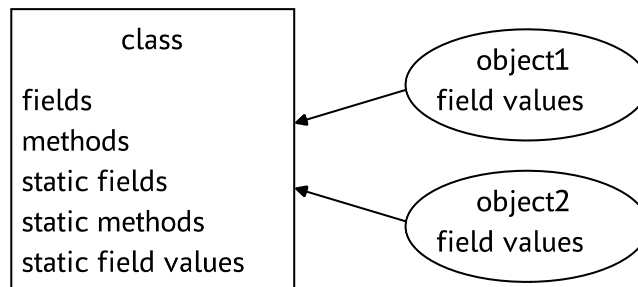


Рис. 1.18: Логическая структура отношения класс-объект

Аналогично статическим методам, **статические поля** принадлежат классу и совершенно ничего «не знают» об объектах.

Важной отличительной чертой статических полей является то, что их значения также хранятся в классе, в отличие от обычных полей, чьи значения хранятся в объектах.

Рисунок 1.18 именно в этом виде автор настоятельно рекомендует если не заучить, то хотя бы хорошо запомнить, он ещё пригодится в дальнейшем обучении и работе. Из этого же изображения можно сделать несколько выводов.

```
public class Cat {
   static int pawsCount = 4;

   String name;
   String color;
   int age;

   // ...
}
```

Помимо того, что статические поля – это полезный инструмент создания общих свойств, это ещё и опасный инструмент создания общих свойств. Так, например, мы знаем, что у котов четыре лапы, а не шесть и не восемь. Не создавая никакого Барсика будет понятно, что у кота – четыре лапы. Это полезное поведение любого класса и его объектов.

```java
public class Cat {
    static int pawsCount = 4;

    static String name;
    String color;
    int age;

    // ...
}
```

У каждого кота есть имя, и коты хранят значение своего имени каждый сам у себя. А знают экземпляры о названии поля потому что знают, какого класса они экземпляры. В чём опасность? Что если по невнимательности добавить свойство статичности к имени кота?

Создав тех же котов, такой класс вернёт двух Мурзиков и ни одного Барсика. Почему это произошло? По факту переменная одна на всех, и значение тоже одно, а значит при каждом вызове конструктора меняется именно оно, а все остальные коты, ничего не подозревая, смотрят на значение общей переменной и возвращают его. Поэтому, к статическим переменным, как правило, обращаются не по ссылке на объект – `cat1.name`, а по имени класса – `Cat.name`.

Статические переменные – редкость в Java. Вместо них применяют статические константы. Они определяются ключевыми словами `static final` и по соглашению о внешнем виде кода пишутся в верхнем регистре, разделяя слова символом нижнего подчёркивания, так называемым UPPER_SNAKE_CASE.

Задание для самопроверки

1. Что такое класс?
2. Что такое поле класса?
3. На какие три этапа делится создание объекта?
4. Какое свойство добавляет ключевое слово static полю или методу?
   (a) неизменяемость;
   (b) принадлежность классу;
   (c) принадлежность приложению.
5. Может ли статический метод получить доступ к полям объекта?
   (a) не может;
   (b) может только к константным;
   (c) может только к неинициализированным.

1.3.2. Устройство памяти. Стек, куча и сборка мусора

Это погружение в управление памятью Java позволит расширить ваши знания о том, как работает куча, ссылочные типы и сборка мусора, понять глубинные процессы и, как следствие, писать более хорошие программы. Для оптимальной работы приложения JVM делит память на область стека (stack) и область кучи (heap). Всякий раз, когда объявляются новые переменные, создаются объекты или вызывается новый метод, JVM выделяет память для этих операций в стеке или в куче. На рисунке 1.19 представлена общая модель организации памяти в Java.

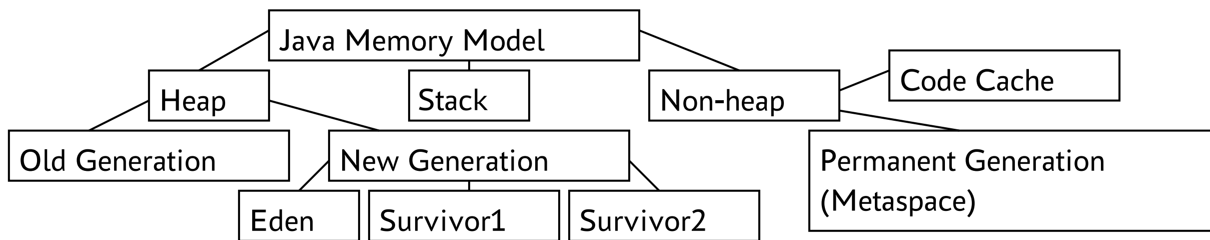


Рис. 1.19: Устройство памяти

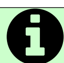
Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи. Данная память в Java работает по схеме LIFO (last in, first out, последний зашел, первый вышел).

Немного забегая вперёд можно сказать, что все потоки, работающие в JVM, имеют свой стек. Пока достаточно отождествлять поток и собственно исполнение программы. Стек в свою очередь держит информацию о том, какие методы вызвал поток. Назовём это «стеком вызовов». Стек вызовов возобновляется, как только поток завершает выполнение своего кода. Каждый слой стека вызовов содержит все локальные переменные для вызываемого метода и потока. Все локальные переменные примитивных типов полностью хранятся в стеке потоков и не видны другим потокам.

Особенности стека:
- Он заполняется и освобождается по мере вызова и завершения новых методов;
- Переменные на стеке существуют до тех пор, пока выполняется метод в котором они были созданы;
- Если память стека будет заполнена, Java бросит исключение `java.lang.StackOverflowError`;
- Доступ к этой области памяти осуществляется быстрее, чем к куче;
- Является потокобезопасным, поскольку для каждого потока создается свой отдельный стек.

Куча содержит все объекты, созданные в приложении, независимо от того, какой поток создал объект. Неважно, был ли объект создан и присвоен локальной переменной или создан как переменная-член другого объекта, он хранится в куче.

Локальная переменная может быть примитивной, но также может быть ссылкой на объект. В этом случае ссылка (локальная переменная) хранится на стеке, но сам объект хранится в куче. Объект использует методы, эти методы содержат локальные переменные. Эти локальные переменные (то есть в момент выполнения метода) также хранятся на стеке, несмотря на то, что объект, который использует метод, хранится в куче. Переменные-члены класса хранятся в куче вместе с самим классом. Это верно как в случае, когда переменная-член имеет примитивный тип, так и в том случае, если она является ссылкой на объект. Статические переменные класса также хранятся в куче вместе с определением класса.

 В общем случае, эти объекты имеют глобальный доступ и могут быть получены из любого места программы.

Куча разбита на несколько более мелких частей, называемых поколениями:
- Young Generation — область где размещаются недавно созданные объекты. Когда она заполняется, происходит быстрая сборка мусора;
- Old (Tenured) Generation — здесь хранятся долгоживущие объекты. Когда объекты из Young Generation достигают определенного порога «возраста», они перемещаются в Old Generation;

– Permanent Generation — эта область содержит метаинформацию о классах и методах приложения, но начиная с Java 8 данная область памяти была упразднена. В Java 8 Permanent Generation заменён на Metaspace - его динамически изменяемый по размеру аналог. Именно здесь находятся статические поля.

Особенности кучи:

– В общем случае, размеры кучи на порядок больше размеров стека
– Когда эта область памяти полностью заполняется, Java бросает `java.lang.OutOfMemoryError`;
– Доступ к ней медленнее, чем к стеку;
– Эта память, в отличие от стека, автоматически не освобождается. Для сбора неиспользуемых объектов используется сборщик мусора;
– В отличие от стека, который создаётся для каждого потока свой, куча не является потокобезопасной, поскольку для всех одна, и ее необходимо контролировать, правильно синхронизируя код.

Также, хотелось бы отметить, что мы можем использовать -Xms и -Xmx опции JVM, чтобы определить начальный и максимальный размер памяти в куче. Для стека определить размер памяти можно с помощью опции -Xss;

Управление неиспользуемыми объектами

– запускается автоматически Java, и Java решает, запускать или нет этот процесс;
– это дорогостоящий процесс. При запуске сборщика мусора все потоки в приложении приостанавливаются (в зависимости от типа GC);
– гораздо более сложный процесс, чем просто сбор мусора и освобождение памяти.

Программисту доступен метод класса System, который мы можем явно вызвать и ожидать, что сборщик мусора будет запускаться при выполнении этой строки кода. Это ошибочное предположение. Java решать, делать это или нет. Явно вызывать `System.gc()` не рекомендуется.

Поскольку это довольно сложный процесс и может повлиять на производительность всего приложения, он реализован весьма разумно. Для этого используется так называемый процесс «Mark and Sweep» (отмечай и подметай). Java анализирует переменные из стека и «отмечает» все объекты, которые необходимо поддерживать в рабочем состоянии. Затем все неиспользуемые объекты очищаются. Фактически, чем больше мусора и чем меньше объектов помечены как живые, тем быстрее идет процесс. Чтобы сделать это еще более оптимизированным, память кучи состоит из нескольких частей.

1. Молодое поколение – Все новые объекты начинаются с молодого поколения. Как только они выделены в коде Java, они попадают в этот подраздел, называемый **eden space**. В конце концов пространство эдема заполняется объектами. На этом этапе происходит незначительная сборка мусора, так называемая minor collection. Некоторые объекты (те, на которые есть ссылки) помечаются, а некоторые (те, на которые нет ссылок) - нет. Те, которые были отмечены, затем переходят в другой подраздел молодого поколения под названием пространство выживших (само пространство выживших разделено на две части). Те, которые остались немаркированными, удаляются автоматической сборкой мусора.
2. Выжившее поколение – Так будет продолжаться до тех пор, пока пространство eden снова не заполнится; на этом этапе начинается новый цикл. События minor collection повторяются, но в этом цикле все отмеченные объекты, которые выживают как из пространства eden, так и из S0, фактически попадают во вторую часть пространства survivor, называемую S1.

3. Третье поколение – Любые объекты, попадающие в пространство выживших, помечаются счетчиком возраста. Алгоритм проверяет этот счётчик, чтобы увидеть, соответствует ли он пороговому значению для перехода в старое поколение. Главная мысль в том, что объекты не обязательно переходят из S0 в S1 пространства выживших. На самом деле, они просто чередуются с тем, куда они переключаются при каждой minor сборке мусора. Если эти процессы обобщить, то все новые объекты начинаются в пространстве eden, а затем в конечном итоге попадают в пространство survivor, поскольку они переживают несколько циклов сборки мусора.
4. Старое поколение можно рассматривать как место, где лежат долгоживущие объекты. Когда объекты собирают мусор из старого поколения, происходит крупное событие сборки мусора. Старое поколение состоит только из одной секции, называемой постоянным поколением.
5. Постоянное поколение – Постоянное поколение не заполняется, когда объекты старого поколения достигают определенного порога, а затем перемещаются (повышаются) в постоянное поколение. Скорее, постоянное поколение немедленно заполняется JVM метаданными, которые представляют классы и методы приложений во время выполнения. JVM иногда может следовать определенным правилам для очистки постоянного поколения, и когда это происходит, это называется полной сборкой мусора major collection.

Также, хотелось бы ещё раз упомянуть событие под названием остановить мир. Когда происходит небольшая сборка мусора (для молодого поколения) или крупная сборка мусора (для старого поколения), мир останавливается; другими словами, все потоки приложений полностью останавливаются и должны ждать завершения события сборки мусора.

Сборщик мусора. Реализации

1. Последовательный сборщик мусора. Это самая простая реализация GC, поскольку она в основном работает с одним потоком. В результате эта реализация GC замораживает все потоки приложения при запуске. Поэтому не рекомендуется использовать его в многопоточных приложениях, таких как серверные среды;
2. Параллельный сборщик мусора. Это GC по умолчанию для JVM, который иногда называют сборщиками пропускной способности. В отличие от последовательного сборщика мусора, он использует несколько потоков для управления пространством кучи, но также замораживает другие потоки приложений во время выполнения GC. Если мы используем этот GC, мы можем указать максимальные потоки сборки мусора и время паузы, пропускную способность и занимаемую площадь (размер кучи);
3. Сборщик мусора CMS. Реализация Concurrent Mark Sweep (CMS) использует несколько потоков сборщика мусора для сбора мусора. Он предназначен для приложений, которые требуют более коротких пауз при сборке мусора и могут позволить себе совместно использовать ресурсы процессора со сборщиком мусора во время работы приложения. Проще говоря, приложения, использующие этот тип GC, в среднем работают медленнее, но не перестают отвечать, чтобы выполнить сборку мусора.

Следует отметить, что, поскольку этот GC является параллельным, вызов явной сборки мусора, такой как использование System.gc() во время работы параллельного процесса, приведет к сбою или прерыванию параллельного режима;

4. Сборщик мусора G1. Сборщик мусора G1 (Garbage First) предназначен для приложений, работающих на многопроцессорных компьютерах с большим объемом памяти. Он доступен с обновления 4 JDK7 и в более поздних версиях. Сборщик G1 заменит сборщик CMS, поскольку он более эффективен;

5. Z сборщик мусора. ZGC (Z Garbage Collector) - это масштабируемый сборщик мусора с низкой задержкой, который дебютировал в Java 11 в качестве экспериментального варианта для Linux. JDK 14 представил ZGC под операционными системами Windows и macOS. ZGC получил статус production начиная с Java 15.

Итоги рассмотрения устройства памяти

- куча доступна везде, объекты доступны отовсюду
- все объекты хранятся в куче, все локальные переменные хранятся на стеке
- стек недолговечен
- и стек и куча могут быть переполнены
- куча много больше стека, но стек гораздо быстрее

Задания для самопроверки

1. По какому принципу работает стек?
2. Что быстрее, стек или куча?
3. Что больше, стек или куча?

1.3.3. Конструкторы

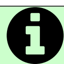
Контроль над созданием объекта

Чтобы создать объект мы тратим одну строку кода `Cat cat1 = new Cat();` поля этого объекта заполнятся автоматически значениями по-умолчанию (числовые - 0, логические - `false`, ссылочные - `null`). Часто нужно при создании дать коту какое-то имя, указать его возраст и цвет, поэтому пишем ещё три строки кода.

В таком подходе есть несколько недостатков:

1. прямое обращение к полям объекта,
2. если полей у класса будет намного больше, то для создания всего лишь одного объекта будет уходить 5-10-15 строк кода, что очень громоздко и утомительно.

Было бы неплохо иметь возможность сразу, при создании объекта указывать значения его полей. Для инициализации объектов при создании в Java предназначены конструкторы.



Конструктор - это частный случай метода в том смысле, что он тоже выполняет какие-то действия. Имя конструктора обязательно должно совпадать с именем класса, возвращаемое значение не пишется.

Если создать конструктор класса Cat, как показано в листинге 1.3, он автоматически будет вызываться при создании объекта. Теперь, при создании объектов класса Cat, все коты будут иметь одинаковые имена, цвет и возраст (это будут белые двухлетние Барсики).

Листинг 1.3: Не самый лучший конструктор

```
1 public class Cat {
2     String name;
3     String color;
```

```
    int age;

    public Cat() {
        name = "Barsik";
        color = "White";
        age = 2;
    }

    // ...
}
```

При использовании такого конструктора, все созданные коты будут одинаковыми, пока мы вручную не поменяем значения их полей. Чтобы можно было указывать начальные значения полей объектов необходимо создать параметризированный конструктор.

Листинг 1.4: Параметризированный конструктор

```
public class Cat {
    String name;
    String color;
    int age;

    Cat(String n, String c, int a) {
        name = n;
        color = c;
        age = a;
    }

    // ...
}
```

В приведенном примере, в параметрах конструктора используется первая буква от названия поля, это сделано для упрощения понимания логики заполнения полей объекта, и будет заменено на более корректное использование ключевого слова `this`. При такой форме конструктора, когда появится необходимость создавать в программе кота, необходимо будет обязательно указать его имя, цвет и возраст. Набор полей, которые будут заполнены через конструктор, определяется разработчиком класса сами, то есть язык не обязывает заполнять все поля, которые есть в классе записывать в параметры конструктора, но при вызове обязательно заполнить аргументами все, что есть в параметрах, как при вызове метода.

```
Cat cat1 = new Cat("Barsik", "White", 4);
Cat cat2 = new Cat("Murzik", "Black", 6);
```

Наборы значений имён, цветов и возрастов будут переданы в качестве аргументов конструктора (`n`, `c`, `a`), а конструктор уже перезапишет полученные значения в поля объект (`name`, `color`, `age`). То есть начальные значения полей каждого из объектов будут определяться тем, что мы передадим ему в конструкторе.

Язык позволяет как не объявлять ни одного конструктора, так и объявить их несколько. Также как и при перегрузке методов, имеет значение набор аргументов, не может быть нескольких конструкторов с одинаковым набором аргументов. Соответствующий перегружаемый конструктор вызывается в зависимости от аргументов, указываемых при выполнении оператора new.

Как только в программе в классе создана своя реализация конструктора, пустой конструктор, называемый также **конструктором по-умолчанию** автоматически создаваться не будет. И если понадобится такая форма конструктора, необходимо будет создать его вручную, `public Cat() {}`.

То есть, компилятор как-бы думает, что если вы не описали конструкторы, значит вам не важно, как будет создаваться объект, а значит, хватит пустого конструктора, ну а если конструктор написан, значит выкручивайтесь сами.

Ключевое слово this

В контексте конструкторов, применять this нужно в двух случаях:

1. Когда у переменной экземпляра класса и переменной метода/конструктора одинаковые имена;
2. Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора.

Внимательно посмотрев на параметризированный конструктор (листинг 1.4), видим, что переменные в параметрах называются не также, как поля класса.

Нельзя просто сделать названия параметров идентичными названиям полей, в этом случае возникает проблема. Для примера возьмём имя кота, поле `String name`. Один `String name` принадлежит классу `Cat`, а другой `String name` находится в локальной видимости конструктора. JVM, как и любой другой электрический прибор всегда идёт по пути наименьшего сопротивления, когда есть неопределённость. То есть, когда написана строка `name = name;` Java берёт самую близкую `name` из конструктора и для левой и для правой части оператора присваивания, что не имеет никакого смысла.

Листинг 1.5: Использование ключевого слова `this` для параметров

```
public class Cat {
   String name;
   String color;
   int age;

   public Cat(String name, String color, int age) {
      this.name = name;
      this.color = color;
      this.age = age;
   }

   // ...
}
```

Ключевое слово `this` сошлётся на вызвавший объект, в результате чего (в листинге 1.5)имя котика через конструктор будет установлено создаваемому объекту. Таким образом, здесь `this` позволяет не вводить новые переменные для обозначения одного и того же, что позволяет сделать код менее перегруженным дополнительными переменными.

Второй случай частого использования this с конструкторами - вызов одного конструктора из другого. это может пригодиться когда в классе описано несколько конструкторов и не хочется в новом конструкторе переписывать код инициализации, приведенный в конструкторе ранее[5]. В листинге 1.5 вызывается обычный конструктор с тремя параметрами, который принимает имя цвет и возраст, но, допустим, когда котята рождаются возраст им задавать смысла нет, поэтому, может пригодиться и конструктор просто с именем и цветом, а зачем писать присваивание имени и цвета несколько раз, если можно вызвать соответствующий конструктор?

Листинг 1.6: Использование ключевого слова this для конструктора

```java
public class Cat {
   String name;
   String color;
   int age;

   public Cat(String name, String color) {
      this.name = name;
      this.color = color;
   }

   public Cat(String name, String color, int age) {
      this(name, color);
      this.age = age;
   }

   // ...
}
```

На такой вызов есть ограничение, конструктор из конструктора можно вызвать только один раз и только на первой строке конструктора.

Ключевое слово this в Java используется только в составе экземпляра класса. Но неявно ключевое слово this передается во все методы, кроме статических (поэтому this часто называют неявным параметром) и может быть использовано для обращения к объекту, вызвавшему метод.

Существует ещё один вид конструктора - это **конструктор копирования**. Чтобы создать конструктор копирования, возможно объявить конструктор, который принимает объект того же типа, в нашем случае котика, в качестве параметра, а в самом конструкторе аналогично конструктору, заполняющему все параметры, заполнить каждое поле входного объекта в новый экземпляр.

```java
public Cat (Cat cat) {
   this(cat.name, cat.color, cat.age);
}
```

Благодаря имеющемуся конструктору со всеми нужными параметрами, с помощью ключевого слова this явно вызывается конструктор заполняющий все поля создаваемого кота,

[5] один из базовых принципов программирования - DRY (от англ dry - чистый, сухой, акроним don't repeat yourself) - не повторяйся. Его антагонист WET (от англ wet - влажный, акроним write everything twice) - пиши всё дважды.

значениями из переданного объекта, фактически, его копирующий. То, что мы имеем здесь, – это неглубокая копия. Если класс имеет изменяемые поля, например, массивы, то мы можем вместо простой сделать глубокую копию внутри его конструктора копирования. При глубокой копии вновь созданный объект не должен зависеть от исходного, а значит просто скопировать ссылку на массив будет недостаточно

Задания для самопроверки

1. Для инициализации нового объекта абсолютно идентичными значениями свойств переданного объекта используется
   (a) пустой конструктор
   (b) конструктор по-умолчанию
   (c) конструктор копирования
2. Что означает ключевое слово this?

1.3.4. Инкапсуляция

Инкапсуляция связывает данные с манипулирующим ими кодом и позволяет управлять доступом к членам класса из отдельных частей программы, предоставляя доступ только с помощью определенного ряда методов, что позволяет предотвратить злоупотребление этими данными. То есть класс должен представлять собой «черный ящик», которым возможно пользоваться, но его внутренний механизм защищен от повреждений.

Инкапсуляция - (англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.

В Java в роли чёрного ящика выступает класс. Класс содержит в себе и данные (поля класса), и действия (методы класса) для работы с этими данными. Все члены класса в языке Java - поля и методы - имеют модификаторы доступа. Ранее уже было описан модификатор `public`, означающий доступность отовсюду, обычно используется для методов.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

Есть два модификатора, которые уже известны и активно используются, это `public` и package-private, также известный как default, пакетный или отсутствующий модификатор. Что это значит? Это значит, что вообще всё что пишется в Java имеет уровень доступа, и если этот уровень не определён явно, то Java отнесёт данные к уровню доступности внутри пакета.

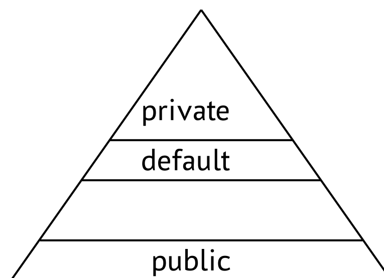


Рис. 1.20: Модификаторы доступа и их относительная область видимости

Модификатор `private` определяет доступность только внутри класса, и предпочтительнее всех.

Внимательно рассмотрим класс котика из листинга 1.5. Например, кто-то может создать хорошего кота, а потом его переименовать, перекрасить в зелёный цвет или сделать ему отрицательный возраст, в результате в программе находятся объекты с некорректным состоянием. Все поля находятся в пакетном доступе. К ним можно обратиться в любом месте пакета: достаточно просто создать объект.

`private` — самый строгий модификатор доступа в Java. Если его использовать, поля класса не будут доступны за его пределами. Решая проблему несанкционированного доступа была получена проблема штатного функционирования, доступ к полям закрыт, в программе нельзя даже получить вес существующей кошки, если это понадобится.

Необходимо решить вопросы с получением и изменением значений полей. На помощь приходят «геттеры» и «сеттеры». Название происходит от английского «get» — «получать» (т.е. «метод для получения значения поля») и «set» — «устанавливать».

Листинг 1.7: Геттеры и сеттеры для всех полей

```
public class Cat {
   private String name;
   private String color;
   private int age;

   // ...

   public String getName() {
     return name;
   }
   public String getColor() {
     return color;
   }
   public int getAge() {
     return age;
   }
   public void setName(String name) {
     this.name = name;
   }
   public void setColor(String color) {
     this.color = color;
   }
   public void setAge(int age) {
     this.age = age;
```

```
25      }
26  }
```

В листинге 1.7 показан пример написания геттеров и сеттеров для всех полей класса, что даёт возможность получать данные и устанавливать их. Например, с помощью `getColor` возможно получить текущее значение окраса котика.

Важно, что создавая для класса геттеры и сеттеры не только появляется возможность дополнять установке и возвращению значений полей дополнительную логику, но и возможность регулировать доступ к полям. Например, если в программе нужно запретить менять котикам окрас, то для класса просто не пишется соответствующий сеттер.

Внимательно осмотрев класс кота возможно прийти к выводу, что хранить возраст котов очень неудобно, потому что каждый год нужно будет обновлять это значение для каждого объекта кота в программе, а это может оказаться утомительно. Выходом может оказаться хранение не возраста, а неизменяемого параметра - даты рождения и подсчёт возраста каждый раз, когда его запрашивают, ведь человеку, который запрашивает возраст кота, не интересно, каким образом получено значение, прочитано из поля или вычислено, ему важен конечный результат. Это и есть инкапсуляция, сокрытие реализации.

Задания для самопроверки

1. Перечислите модификаторы доступа
2. Инкапсуляция - это
   (a) архивирование проекта
   (b) сокрытие информации о классе
   (c) создание микросервисной архитектуры

1.3.5. Наследование

Проблема

Второй кит ООП после инкапсуляции - наследование.

Представим, что есть необходимость создать помимо класса котиков, класс собачек. Данный класс будет выглядеть очень похожим образом, только он будет не мяукать, а гавкать, и заменим обоим животным прыжок на простое перемещение на лапках.

Листинг 1.8: Класс кота

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String
        color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getColor() {
        return color;
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setColor(String color)
        {
        this.color = color;
    }

    public void setAge(int age) {
        this.age = age;
    }

    void voice() {
        System.out.println(name + "
            meows");
    }

    void move() {
        System.out.println(name + "
            walks on paws");
    }
}
```

Листинг 1.9: Класс собаки

```
public class Dog {
    private String name;
    private String color;
    private int age;

    public Dog(String name, String
        color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getColor() {
        return color;
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setColor(String color)
        {
        this.color = color;
    }

    public void setAge(int age) {
        this.age = age;
    }

    void voice() {
        System.out.println(name + "
            barks");
    }

    void move() {
        System.out.println(name + "
            walks on paws");
    }
}
```

Очевидно это не DRY и неприемлемо, если появится необходимость описать классы для целого зоопарка. В приведённых классах есть очень много абсолютно одинаковых и очень похожих полей и методов.

> Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Наследование в Java реализуется ключевым словом `extends` (англ. - расширять). И кот и пёс являются животными, у всех описываемых в программе животных есть имя, возраст, окрас, все описываемые животные могут бегать, прыгать, и откликаться на имя. Создав так называемый **родительский класс**, или суперкласс (листинг 1.10), и поместив в него поля, геттеры и сеттеры, стало возможным убрать поля, геттеры и сеттеры из кота и пса. Если полей много, лаконичность описания родственных классов может быть весьма ощутимой.

Листинг 1.10: Класс животного

```
public class Animal {
   private String name;
   private String color;
   private int age;

   public String getName() {
     return name;
   }

   public String getColor() {
     return color;
   }

   public int getAge() {
     return age;
   }

   public void setName(String name) {
     this.name = name;
   }

   public void setColor(String color) {
     this.color = color;
   }

   public void setAge(int age) {
     this.age = age;
   }

}
```

Чтобы унаследовать один класс от другого, нужно после объявления указать ключевое слово `extends` и написать имя родительского класса как это показано в листингах 1.11 и 1.12. Если перенос геттеров возможен, то значит достаточно безболезненно можно перенести и одинаковые методы.

Простой перенос кода в родительский класс показал наличие проблемы. Модификатор `private` определяет область видимости только внутри класса, а если нужно чтобы перемен-

ную было видно ещё и в классах-наследниках, нужен хотя бы модификатор доступа по умолчанию. Если же класс наследник создаётся в каком-то другом пакете, то и default не подойдёт.

Листинг 1.11: Класс собаки

```
public class Dog extends Animal {
    public Dog(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    void voice() {
        System.out.println(name + " barks");
    }

    void move() {
        System.out.println(name + " walks on paws");
    }
}
```

Листинг 1.12: Класс кота

```
public class Cat extends Animal {
    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    void voice() {
        System.out.println(name + " meows");
    }

    void move() {
        System.out.println(name + " walks on paws");
    }
}
```

То есть, к членам данных и методам класса можно применять следующие модификаторы доступа

- `private` - содержимое класса доступно только из методов данного класса;
- `public` - есть доступ фактически отовсюду;
- default (по-умолчанию) - содержимое класса доступно из любого места пакета, в котором этот класс находится;
- `protected` (защищенный доступ) содержимое доступно также как с модификатором по-умолчанию, но ещё и для классов-наследников.

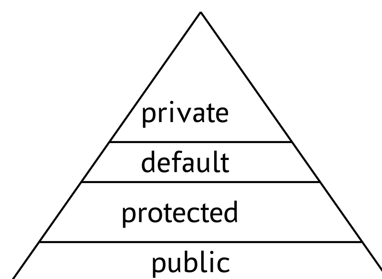
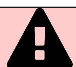


Рис. 1.21: Модификаторы доступа и их относительная область видимости

То есть верным вариантом в листинге 1.10 будет применение модификатора `protected`.

Конструкторы в наследовании



Несмотря на то, что конструктор - это частный случай метода, если перенести одинаковые конструкторы кота и пса в общий класс животного, программа снова перестанет работать, потому что важно учитывать механику вызова конструкторов при наследовании.

Важно запомнить, что при создании любого объекта в первую очередь вызывается конструктор его базового (родительского) класса, а только потом — конструктор самого класса, объект которого мы создаем. То есть при создании объекта `Cat` сначала отработает конструктор класса `Animal`, а только потом конструктор `Cat`. Но, поскольку конструктор по-умолчанию в нашем случае перестал создаваться, а других может быть бесконечно много, это создало неопределённость, которую программа разрешить не может.

При описании класса, можно явно вызвать конструктор базового класса в конструкторе класса-потомка. Базовый класс еще называют «суперклассом», поэтому в Java для его обозначения используется ключевое слово `super`. Здесь такое же ограничение, как и при вызове конструкторов данного класса (через `this`) - вызов такого конструктора может быть только один и быть только первой строкой. Таким образом, код, для всех животных в программе будет выглядеть следующим образом:

Листинг 1.13: Класс животного

```java
public class Animal {
    private String name;
    private String color;
    private int age;

    public Animal(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public String getName() {
      return name;
    }

    public String getColor() {
      return color;
    }

    public int getAge() {
      return age;
    }

    public void setName(String name) {
      this.name = name;
    }

    public void setColor(String color) {
      this.color = color;
    }

    public void setAge(int age) {
      this.age = age;
    }

    void move() {
        System.out.println(name + " walks on paws");
    }
}
```

Листинг 1.14: Класс кота

```java
public class Cat extends Animal {
    public Cat(String name, String color, int age) {
        super(name, color, age);
    }

    void voice() {
        System.out.println(name + " meows");
    }
}
```

Листинг 1.15: Класс собаки

```java
public class Dog extends Animal {
    public Dog(String name, String color, int age) {
        super(name, color, age);
    }

    void voice() {
        System.out.println(name + " barks");
    }
}
```

Листинг 1.16: Класс птицы

```java
public class Bird extends Animal {
    private int flyHeight;

    public Bird(String name, String color, int age, int flyHeight) {
        super(name, color, age);
        this.flyHeight = flyHeight;
    }

    void voice() {
        System.out.println(name + " tweets");
    }

    void fly() {
        System.out.println(name + " flies at " + flyHeight + " m");
    }
}
```

Для примера, создали ещё один класс, наследника животного, чтобы использовать наследование по назначению. Наследование реализуется через ключевое слово `extends`, расширять. Важно, что класс-родитель расширяется функциональностью или свойствами класса-наследника. Это позволило, например, добавить в птичку такое свойство как высота полёта

и такой метод как летать, в дополнение к тому, что умеют все животные.

Object и каскадное наследование

Множественное наследование запрещено! Для каждого создаваемого подкласса можно указать только один суперкласс. В Java не поддерживается множественное наследование, то есть наследование одного класса от нескольких суперклассов. Зато возможно каскадное наследование, то есть класс-наследник вполне может быть чьим-то родителем.

Если класс-родитель не указан, таковым считается класс Object. Таким образом можно сделать вывод о том, что любой класс в джава так или иначе - наследник Object и, соответственно, всех его свойств и методов. Объект подкласса представляет объект суперкласса, выражаясь проще, возможно ко всем котикам обращаться через общее название Животное, и ко всем объектам в программе возможно обратиться через класс Object. Поэтому в программе не будет ошибкой написать подобный код:

```
Object animal = new Animal("Cat", "Black", 3);
Object cat = new Cat("Murka", "Black", 4);
Object dog = new Dog("Bobik", "White", 2);
Animal dogAnimal = new Bird("Chijik", "Grey", 3, 10);
Animal catAnimal = new Cat("Marusya", "Orange", 1);
```

Это так называемое восходящее преобразование (от подкласса внизу к суперклассу вверху иерархии) или **upcasting**. Такое преобразование осуществляется автоматически. Обратное не всегда верно. Например, объект Animal не всегда является объектом Cat или Dog. Поэтому нисходящее преобразование или downcasting от суперкласса к подклассу автоматически не выполняется. В этом случае необходимо использовать операцию преобразования типов.

```
Object animal = new Cat("Murka", "Black", 4);
Cat cat = (Cat)animal;
cat.move();
```

Обратите внимание, что в данном случае переменная animal приводится к типу Cat. И затем через объект cat становится возможным обратиться к функционалу кота. Важно при этом, что изначально оператором new был создан объект кота, а не Object или Animal/

Оператор instanceof и ключевое слово final

Оператор instanceof возвращает истину, если объект принадлежит классу или его суперклассам и ложь в противном случае. Нередко данные приходят извне, и невозможно точно знать, какой именно объект эти данные представляют. Возникает большая вероятность столкнуться с ошибкой преобразования типов. И перед тем, как провести преобразование типов, необходимо проверить возможность выполнить приведение с помощью оператора instanceof.

```
Object cat = new Cat("Murka", "Black", 4);
if (cat instanceof Dog) {
  Dog dogIsCat = (Dog) cat;
  dogIsCat.voice();
```

```
} else {
    System.out.println("Conversion is invalid");
}
```

Выражение `cat instanceof Dog` проверяет, является ли переменная `cat` объектом типа `Dog`. Так как в данном случае явно кот не является собакой, то такая проверка вернет значение `false`, и преобразование не сработает. А выражение `cat instanceof Animal` выдало бы истинный результат.

Ключевое слово `final`. Класс с конечной реализацией.

Ключевое слово final может применяться к классам, методам, переменным (в том числе аргументам методов). Применительно к классам, это возможность запретить наследование. То есть, если пометить этим ключевым словом, например, птичку, тогда, если в коде начать писать класс например, попугайчика, и указать, наследование от птицы, то выведется `Cannot inherit from final`.

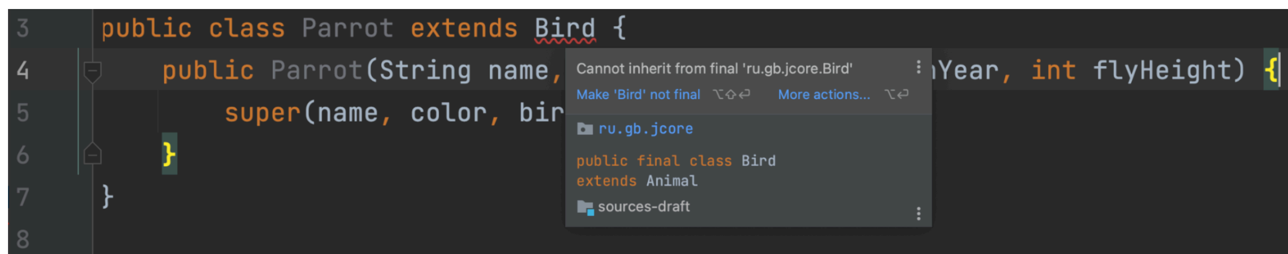


Рис. 1.22: Ошибка наследования от final класса

Абстракция

Иногда абстракцию выделяют, как четвёртый принцип ООП.

Абстрактный класс — это написанная максимально широкими мазками, приблизительная «заготовка» для группы будущих классов. Эту заготовку нельзя использовать в чистом виде — слишком «сырая». Но она описывает некое общее состояние и поведение, которым будут обладать будущие классы — наследники абстрактного класса.

Абстрактными могут быть не только классы, но и методы. **Абстрактный метод -** это метод без реализации. Все животные в примерах выше умеют издавать свой звук. Известно, на этапе проектирования животного, что все животные должны издавать звук, но невозможно сказать, какой именно. Поэтому, определяется, что у животного есть метод издать звук, но реализация этого метода в животном не пишется, слишком мало сведений. Поэтому, метод помечается как абстрактный.

Что будет, если программа попытается вызвать метод `voice()` у животного?

Класс животного максимально абстрактно описывает нужную нам сущность — животное. Но в мире не существует «просто животных». Есть губки, иглокожие, хордовые и т.д. Данный класс теперь слишком абстрактный, чтобы программа могла с ним нормально взаимодействовать, а значит просто является чертежом по которому будут создаваться дальнейшие классы животных. Отметим этот факт явно, написав ключевое слово `abstract` у класса.

- Абстрактный метод - это метод не содержащий реализации (объявление метода).
- Абстрактный класс - класс содержащий хотя бы один абстрактный метод.
- Абстрактный класс нельзя инстанциировать (создать экземпляр).

Очевидно, что абстрагирование метода вынуждает абстрагировать класс, но не наоборот, абстрактный класс необязательно должен содержать абстрактные методы, фактически, это просто запрещение создания экземпляров.

Задания для самопроверки

1. Какое ключевое слово используется при наследовании?
   (a) parent
   (b) extends
   (c) как в C++, используется двоеточие
2. super - это
   (a) ссылка на улучшенный класс
   (b) ссылка на расширенный класс
   (c) ссылка на родительский класс
3. Не наследуются от Object
   (a) строки
   (b) потоки ввода-вывода
   (c) ни то ни другое

1.3.6. Полиморфизм

Полиморфизм – это возможность объектов с одинаковой спецификацией иметь различную реализацию (Overriding). Полиморфизм выражается возможностью переопределения поведения суперкласса (часто можно встретить утверждение, что при помощи перегрузки. Основная суть в том, что в классе-родителе имеется некоторый метод, но реализация этого метода разная у каждого класса-наследника, фактически это и есть полиморфизм.

Вынесем последний оставшийся в котиках и птичках метод в общий класс животного. В классах-потомках определим такие же методы, как и объявленный метод класса родителя, который хотим изменить.

Листинг 1.17: Класс кота

```
public abstract class Animal {
    // ...

    void voice();

    // ...
}

public class Cat extends Animal {
    public Cat(String name, String color, int age) {
        super(name, color, age);
    }
```

```
13
14    @Override
15    void voice() {
16       System.out.println(name + " meows");
17    }
18 }
```

Получается, при наследовании от `Animal`, у которого есть метод `voice()`, класс котика сам определяет, какой он издаёт звук.

Аннотации реализуют вспомогательные интерфейсы.

Аннотация `@Override` проверяет, действительно ли метод переопределяется, а не перегружается. Если существует ошибка в сигнатуре метода, то компилятор сразу об этом скажет.

Полиморфизм чаще всего используется когда нужно описать поведение абстрактного класса или назначить разным наследникам разное поведение, одинаково названное в классе родителе. Но есть и ситуации, когда все классы делают что-то одинаково, а один делает это как-то иначе. Продемонстрируем на примере класса `Snake`, змея.

По очевидным причинам змейка не может ходить на лапках. Поэтому это поведение у змейки будет переопределено.

Листинг 1.18: Класс кота

```
1  public class Snake extends Animal {
2     public Snake(String name, String color, int age) {
3        super(name, color, age);
4     }
5
6     @Override
7     void move() {
8        System.out.println(name + " crawls");
9     }
10
11    @Override
12    void voice() {
13       System.out.println(name + " hisses");
14    }
15 }
```

Также, например, можно было создать черепаху, которая не умеет бегать, рыбу, которая не издаёт звуков, слона, который не умеет прыгать, в отличие от остальных, практикующих «среднее» поведение.

Стоит помнить, что переопределять можно только нестатические методы. Статические методы не наследуются в привычном смысле и, следовательно, не переопределяются. Создав в животном и коте статический метод с одинаковой сигнатурой мы сможем наблюдать то, что называется хайдингом, иначе сокрытием или перекрытием. Также, обратите внимание, что попытка написать у этих методов аннотацию `@Override` вызовет ошибку компиляции.

```
1  public class Animal {
2     // ...
3     public static void self() { ... }
```

```
}
public class Cat extends Animal {
    // ...
    public static void self() { ... }
}
```

По коду видно, что класс унаследовался и метод должен переопределиться, внешне если создать `Animal a` и `Cat c`, будет похоже, что так и произошло, но если сделать обе переменные типа `Animal`, очевидно, что поведение изменилось. Статические члены класса относятся к классу, т.е. к типу переменной. Поэтому, логично, что если `Cat` имеет тип `Animal`, то и метод будет вызван у `Animal`, а не у `Cat`.

Полиморфизм в языках программирования и теории типов — способность функции обрабатывать данные разных типов. Выделяют параметрический полиморфизм и ad-hoc-полиморфизм.

Широко распространено определение полиморфизма, приписываемое Бьёрну Страуструпу: «один интерфейс — много реализаций». Полиморфизм - это гораздо более широкое понятие, чем просто переопределение методов, в эту тему завязаны разные интересные теории типов и информации, множество парадигм программирования и другое. С утилитарной точки зрения, остался ещё один вариант, который, тем не менее, не дотягивает до истинного полиморфизма.

К полиморфизму также относится перегрузка методов (Overloading) - использование более одного метода с одним и тем же именем, но с разными параметрами в одном и том же классе или между суперклассом и подклассами.

Перегрузка работает также, как работала без явной привязки кода к парадигме ООП, ничего нового, но для порядка следует создать возможность животным перемещаться не только абстрактно, но и на какое-то конкретное место или на какое-то конкретное количество шагов.

```
    void move() {
        System.out.println(name + " walks on paws");
    }

    void move(String to) {
        System.out.println(name + " moves to " + to);
    }

    void move(int steps) {
        System.out.println(name + " moves " + steps + " steps away");
    }
```

Как видно, методы имеют одинаковые названия, но отличаются по количеству параметров и их типу.

Чтобы стиль вашей программы соответствовал концепции ООП и принципам ООП в Java следуйте следующим советам:

- выделяйте главные характеристики объекта;
- выделяйте общие свойства и поведение и используйте наследование при создании объектов;
- используйте абстрактные типы для описания объектов;
- старайтесь всегда скрывать методы и поля, относящиеся к внутренней реализации класса.

Задания для самопроверки

1. Является ли перегрузка полиморфизмом
   (a) да, это истинный полиморфизм
   (b) да, это часть истинного полиморфизма
   (c) нет, это не полиморфизм
2. Что обязательно для переопределения?
   (a) полное повторение сигнатуры метода
   (b) полное повторение тела метода
   (c) аннотация Override

Практическое задание

1. Написать класс кота так, чтобы каждому объекту кота присваивался личный порядковый целочисленный номер.
2. Написать классы кота, собаки, птицы, наследники животного. У всех есть три действия: бежать, плыть, прыгать. Действия принимают размер препятствия и возвращают булев результат. Три ограничения: высота прыжка, расстояние, которое животное может пробежать, расстояние, которое животное может проплыть. Следует учесть, что коты не любят воду.
3. * Добавить механизм, создающий 25% разброс значений каждого ограничения для каждого объекта.

Термины, определения и сокращения

`finally` — часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.

`throws` — ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.

`throw` — оператор, активирующий (выбрасывающий) объект исключения.

`try...catch` — двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.

BIOS — (англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.

CI — (англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CLI — (англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».

cp1251 — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл — именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

1.4. Специализация: ООП и исключения

В предыдущем разделе

Была рассмотрена реализация объектно-ориентированного программирования в Java. Рассмотрели классы и объекты, а также наследование, полиморфизм и инкапсуляцию. Дополнительно был освещён вопрос устройства памяти.

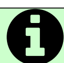
В этом разделе

В дополнение к предыдущему, будут разобраны такие понятия, как внутренние и вложенные классы; процессы создания, использования и расширения перечислений. Более детально будет разобрано понятие исключений и их тесная связь с многопоточностью в Java. Будут рассмотрены исключения с точки зрения ООП, процесс обработки исключений.

- Перечисление;
- Внутренний класс;
- Вложенный класс;
- Локальный класс;
- Исключение;
- Искл. (событие);
- Искл. (объект);
- Обработчик искл.;
- `throw`;
- Stacktrace;
- `try...catch`;
- `throws`;
- `finally`;
- Подавленное искл.;
- Многопоточность;

1.4.1. Перечисления

Кроме восьми примитивных типов данных и классов в Java есть специальный тип, выведенный на уровень синтаксиса языка – `enum` или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора `enum`, после которого идет название перечисления. Затем идет список элементов перечисления через запятую.

 Перечисление – это упоминание объектов, объединённых по какому-либо признаку

Перечисления – это специальные классы, содержащие внутри себя собственные статические экземпляры.

Листинг 1.19: Пример перечисления

```
enum Season { WINTER, SPRING, SUMMER, AUTUMN }
```

Перечисление, фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её. Переменная типа перечисления может хранить любой объект этого исключения.

Листинг 1.20: Переменная типа перечисления

```
Season current = Season.SPRING;
System.out.println(current);
```

Интересно также то, что вывод в терминал и запись в коде у исключений полностью совпадают, поэтому, в результате выполнения этого кода, в терминале будет выведено

```
SPRING
```

Каждое перечисление имеет статический метод `values()`, возвращающий массив всех констант перечисления.

Листинг 1.21: Вывод всех элементов перечисления

```
Season[] seasons = Season.values();
for (Season s : seasons) {
   System.out.printf("s ", s);
}
```

Именно в этом примере используется цикл foreach для прохода по массиву, для лаконичности записи. Данный цикл берёт последовательно каждый элемент перечисления, присваивает ему имя `s` точно также, как это сделано в примере выше, делает эту переменную доступной в теле цикла в рамках одной итерации, на следующей итерации будет взят следующий элемент, и так далее.

```
WINTER, SPRING, SUMMER, AUTUMN
```

Также, в перечисления встроен метод `ordinal()`, возвращающий порядковый номер определенной константы (нумерация начинается с 0). Обратите внимание на синтаксис, метод можно вызвать только у конкретного экземпляра перечисления, а при попытке вызова у самого класса перечисления, ожидаемо компилятор выдаёт ошибку невозможности вызова нестатического метода из статического контекста.

Листинг 1.22: Метод `ordinal()`

```
System.out.println(current.ordinal());

System.out.println(Seasons.ordinal()); // ошибка
```

Такое поведение возможно, только если номер элемента хранится в самом объекте.

В перечислениях можно наблюдать очень примечательный пример инкапсуляции – неизвестно, хранятся ли на самом деле объекты перечисления в виде массива, но можем вызвать метод values() и получить массив всех элементов перечисления. Неизвестно, хранится ли в каждом объекте перечисления его номер, но можем вызвать его метод `ordinal()`.

Раз перечисление – это класс, возможно определять в нём поля, методы, конструкторы и прочее. Перечисление `Color` определяет приватное поле `code` для хранения кода цвета, а с помощью метода `getCode` он возвращается.

Листинг 1.23: Расширение объекта перечисления

```
public class Main {
   enum Color {
      RED("#FF0000"), BLUE("#0000FF"), GREEN("#00FF00");
      private String code;
      Color(String code) {
         this.code = code;
      }

      public String getCode(){ return code;}
   }

   public static void main(String[] args) {
      System.out.println(Color.RED.getCode());
      System.out.println(Color.GREEN.getCode());
   }
}
```

Через конструктор передается значение пользовательского поля.

Конструктор по умолчанию имеет модификатор `private`. Любой другой модификатор будет считаться ошибкой.

Создать константы перечисления с помощью конструктора возможно только внутри самого перечисления. И что косвенно намекает на то, что объекты перечисления это статические объекты внутри самого класса перечисления. Также важно, что механизм описания конструкторов класса работает по той же логике, что и обычные конструкторы, то есть, при описании собственного конструктора, конструктор по-умолчанию перестаёт создаваться автоматически. Таким образом, с объектами перечисления можно работать точно также, как с обычными объектами.

Листинг 1.24: Вывод значений пользовательского поля перечисления

```
for (Color c : Color.values()) {
   System.out.printf("s(s)\n", c, c.getCode());
}
```

```
RED(#FF0000)
BLUE(#0000FF)
GREEN(#00FF00)
```

Задания для самопроверки

1. Перечисления нужны, чтобы:
   (a) вести учёт созданных в программе объектов;
   (b) вести учёт классов в программе;
   (c) вести учёт схожих по смыслу явлений в программе;
2. Перечисление – это:
   (a) массив

   (b) класс
   (c) объект
3. каждый объект в перечислении – это:
   (a) статическое поле
   (b) статический метод
   (c) статический объект

1.4.2. Внутренние и вложенные классы

В Java есть возможность создавать классы внутри других классов, все такие классы разделены на следующие типы:

1. Non-static nested (inner) classes — нестатические вложенные (внутренние) классы;
   — локальные классы (local classes);
   — анонимные классы (anonymous classes);
2. Static nested classes — статические вложенные классы.

Для рассмотрения анонимных классов понадобятся дополнительные знания об интерфейсах, поэтому будут рассмотрены позднее.

Внутренние классы

Листинг 1.25: Вывод значений пользовательского поля перечисления

```
public class Orange {
   public void squeezeJuice() {
      System.out.println("Squeeze juice ...");
   }
   class Juice {
      public void flow() {
         System.out.println("Juice dripped ...");
      }
   }
}
```

Внутренние классы создаются внутри другого класса. Рассмотрим на примере апельсина с реализацией, как это предлагает официальная документация Oracle. В основной программе необходимо создать отдельно апельсин, отдельно его сок через интересную форму вызова конструктора, показанную в листинге 1.26, что позволяет работать как с апельсином, так и его соком по отдельности.

Листинг 1.26: Обычный апельсин Oracle

```
Orange orange = new Orange();
Orange.Juice juice = orange.new Juice();
orange.squeezeJuice();
juice.flow();
```

Важно помнить, что когда в жизни апельсин сдавливается, из него сам по себе течёт сок, а когда апельсин попадает к нам в программу он сразу снабжается соком.

Листинг 1.27: Необычный апельсин GeekBrains

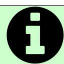
```
public class Orange {
   private Juice juice;
   public Orange() {
      this.juice = new Juice();
   }
   public void squeezeJuice() {
      System.out.println("Squeeze juice ...");
      juice.flow();
   }
   private class Juice {
      public void flow() {
         System.out.println("Juice dripped ...");
      }
   }
}
```

Итак, был создан апельсин, при создании объекта апельсина у него сразу появляется сок. Ниже в классе описано потенциальное наличие у апельсина сока, как его части, поэтому внутри класса апельсин создан класс сока. При создании апельсина создали сок, так или иначе – самостоятельную единицу, обладающую своими свойствами и поведением, отличным от свойств и поведения апельсина, но неразрывно с ним связанную. При попытке выдавить сок у апельсина – объект сока сообщил о том, что начал течь

Листинг 1.28: Использование апельсина GeekBrains

```
Orange orange = new Orange();
orange.squeezeJuice();
```

Таким образом у каждого апельсина будет свой собственный сок, который возможно выжать, сдавив апельсин. В этом смысл внутренних классов не статического типа – нужные методы вызываются у нужных объектов.

 Такая связь объектов и классов называется композицией. Существуют также ассоциация и агрегация.

Если класс полезен только для одного другого класса, то часто бывает удобно встроить его в этот класс и хранить их вместе. Использование внутренних классов увеличивает инкапсуляцию. Оба примера достаточно отличаются реализацией. Пример не из документации подразумевает «более сильную» инкапсуляцию, так как извне ко внутреннему классу доступ получить нельзя, поэтому создание объекта внутреннего класса происходит в конструкторе основного класса – в апельсине. С другой стороны, у примера из документации есть доступ извне ко внутреннему классу сока, но всё равно, только через основной класс апельсина, как и создать объект сока можно только через объект апельсина, то есть подчёркивается взаимодействие на уровне объектов.

Особенности внутренних классов:

– Внутренний объект не существует без внешнего. Это логично – для этого `Juice` был создан внутренним классом, чтобы в программе не появлялись апельсиновые соки из воздуха.
– Внутренний объект имеет доступ ко всему внешнему. Код внутреннего класса имеет доступ ко всем полям и методам экземпляра (и к статическим членам) окружающего класса, включая все члены, даже объявленные как `private`.

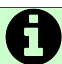
- Внешний объект не имеет доступа ко внутреннему без создания объекта. Это логично, так как экземпляров внутреннего класса может быть создано сколь угодно много, и к какому именно из них обращаться?
- У внутренних классов есть модификаторы доступа. Это влияет на то, где в программе возможно создавать экземпляры внутреннего класса. Единственное сохраняющееся требование — объект внешнего класса тоже обязательно должен существовать и быть видимым.
- Внутренний класс не может называться как внешний, однако, это правило не распространяется ни на поля, ни на методы;
- Во внутреннем классе нельзя иметь не-final статические поля. Статические поля, методы и классы являются конструкциями верхнего уровня, которые не связаны с конкретными объектами, в то время как каждый внутренний класс связан с экземпляром окружающего класса.
- Объект внутреннего класса нельзя создать в статическом методе «внешнего» класса. Это объясняется особенностями устройства внутренних классов. У внутреннего класса могут быть конструкторы с параметрами или только конструктор по умолчанию. Но независимо от этого, когда создаётся объект внутреннего класса, в него неявно передаётся ссылка на объект внешнего класса.
- Со внутренними классами работает наследование и полиморфизм.

Задания для самопроверки

1. Внутренний класс:
   (a) реализует композицию;
   (b) это служебный класс;
   (c) не требует объекта внешнего класса;
2. Инкапсуляция с использованием внутренних классов:
   (a) остаётся неизменной
   (b) увеличивается
   (c) уменьшается
3. Статические поля внутренних классов:
   (a) могут существовать
   (b) могут существовать только константными
   (c) не могут существовать

Локальные классы

Классы – это новый тип данных для программы, поэтому технически возможно создавать классы, а также описывать их, например, внутри методов. Это довольно редко используется но синтаксически язык позволяет это сделать. **Локальные классы** — это подвид внутренних классов. Однако, у локальных классов есть ряд важных особенностей и отличий от внутренних классов. Главное заключается в их объявлении.

 Локальный класс объявляется только в блоке кода. Чаще всего — внутри какого-то метода внешнего класса.

Листинг 1.29: Пример локального класса

```
1 public class Animal {
2     void performBehavior(boolean state) {
```

```
      class Brain {
         void sleep() {
            if (state)
               System.out.println("Sleeping");
            else
               System.out.println("Not sleeping");
         }
      }
      Brain brain = new Brain();
      brain.sleep();
   }
}
```

Например, некоторое животное, у которого устанавливается состояние спит оно или нет. Метод `performBehavior()` принимает на вход булево значение и определяет, спит ли животное. Мог возникнуть вопрос: зачем? Итоговое решение об архитектуре проекта всегда зависит от структуры, сложности и предназначения программы.

Особенности локальных классов:

- Локальный класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода, то есть полям и аргументам метода объявленным как `final`. Начиная с JDK 1.8 локальный класс может обращаться к любым полям и аргументам метода объявленным в текущем блоке кода, даже если они не объявлены как `final`, но только в том случае если их значение не изменяется после инициализации.
- Локальный класс должен иметь свои внутренние копии всех локальных переменных, которые он использует (эти копии автоматически создаются компилятором). Единственный способ обеспечить идентичность значений локальной переменной и ее копии – объявить локальную переменную как `final`.
- Экземпляры локальных классов, как и экземпляры внутренних классов, имеют окружающий экземпляр, ссылка на который неявно передается всем конструкторам локальных классов. То есть, сперва должен быть создан экземпляр внешнего класса, а только затем экземпляр внутреннего класса.

Статические вложенные классы

При объявлении такого класса используется ключевое слово `static`. Для примера в классе котика и заменим метод `voice()` на статический класс.

Листинг 1.30: Статический вложенный класс

```
public class Cat {
   private String name;
   private String color;
   private int age;
   public Cat()
   public Cat(String name, String color, int age) {
      this.name = name;
      this.color = color;
      this.age = age;
   }

```

```java
    static class Voice {
        private final int volume;
        public Voice(int volume) {
            this.volume = volume;
        }
        public void sayMur() {
            System.out.printf("A cat purrs with volume %d\n", volume);
        }
    }
}
```

То есть, такое мурчание котика может присутствовать без видимости и понимания, что именно за котик присутствует в данный момент. Также, добавлена возможность установить уровень громкости мурчанья.

Основное отличие статических и нестатических вложенных классов в том, что объект статического класса не хранит ссылку на конкретный экземпляр внешнего класса.

Без объекта внешнего класса объект внутреннего просто не мог существовать. Для статических вложенных классов это не так. Объект статического вложенного класса может существовать сам по себе. В этом плане статические классы более независимы, чем нестатические. Довольно важный момент заключается в том, что при создании такого объекта нужно указывать название внешнего класса,

Листинг 1.31: Использование статического класса

```java
Cat.Voice voice = new Cat.Voice(100);
voice.sayMur();
```

Статический вложенный класс может обращаться только к статическим полям внешнего класса. При этом неважно, какой модификатор доступа имеет статическая переменная во внешнем классе.

Не следует путать объекты с переменными. Если речь идёт о статических переменных — да, статическая переменная класса существует в единственном экземпляре. Но применительно ко вложенному классу `static` означает лишь то, что его объекты не содержат ссылок на объекты внешнего класса.

Задания для самопроверки

1. Вложенный класс:
   (a) реализует композицию;
   (b) это локальный класс;
   (c) всегда публичный;
2. Статический вложенный класс обладает теми же свойствами, что:
   (a) константный метод
   (b) внутренний класс
   (c) статическое поле

1.4.3. Исключения

Понятие

Язык программирования – это, в первую очередь, набор инструментов. Например, есть художник. У художника есть набор всевозможных красок, кистей, холстов, карандашей, мольберт, ластик и прочие. Это всё его инструменты. Тоже самое для программиста. У программиста есть язык программирования, который предоставляет ему инструменты: циклы, условия, классы, функции, методы, ООП, фрейморки, библиотеки. Исключения – это один из инструментов. Исключения всегда следует рассматривать как ещё один инструмент для работы программиста.

Исключение – это отступление от общего правила, несоответствие обычному порядку вещей

В общем случае, возникновение исключительной ситуации, это ошибка в программе, но основным вопросом является следующий. Возникшая ошибка – это:
- ошибка в коде программы;
- ошибка в действиях пользователя;
- ошибка в аппаратной части компьютера?

Общие сведения

При возникновении ошибок создаётся объект класса «исключение», и в этот объект записывается какое-то максимальное количество информации о том, какая ошибка произошла, чтобы потом прочитать и понять, где проблема. Соответственно эти объекты возможно «ловить и обрабатывать».

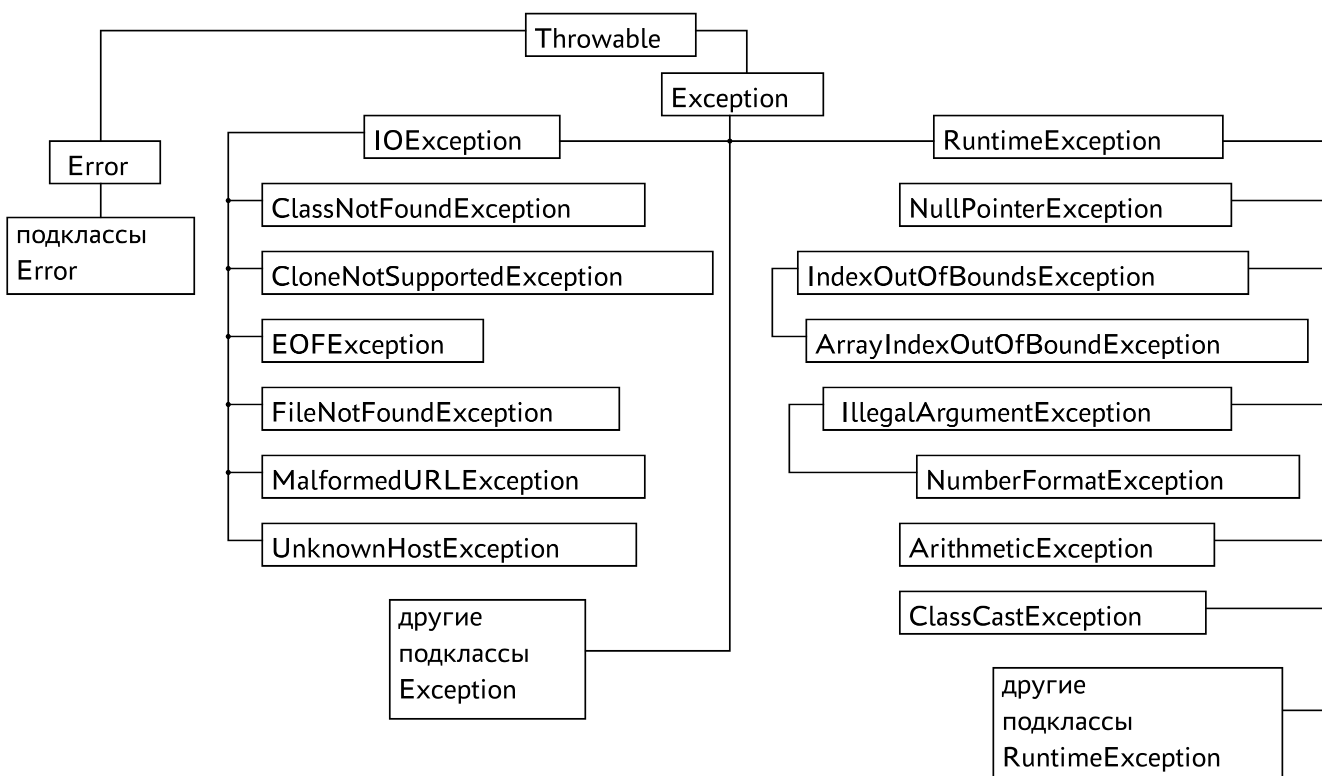


Рис. 1.23: Часть иерархии исключений

Все исключения наследуются от класса `Throwable` и могут быть как обязательные к обработке, так и необязательные. Есть ещё подкласс `Error`, но он больше относится к аппаратным сбоям или серьёзным алгоритмическим или архитектурным ошибкам, и на данном этапе интереса не представляет, потому что поймав, например, `OutOfMemoryError` средствами Java прямо в программе с ним ничего сделать невозможно, такие ошибки необходимо обрабатывать и не допускать в процессе разработки ПО.

Для изучения и примеров, воспользуемся двумя подклассами `Throwable` – `Exception` – `RuntimeException` и `IOException`.

Все исключения (**checked**), кроме наследников `RuntimeException` (**unchecked**), необходимо обрабатывать.

Опишем на простом примере, один метод вызывает другой, второй вызывает третий и последний всё портит:

Листинг 1.32: Цепочка методов

```
private static int div0(int a, int b) {
    return a / b;
}

private static int div1(int a, int b) {
    return div0(a, b);
}

private static int div2(int a, int b) {
    return div1(a, b);
}
```

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ru.gb.jcore.Main.div0(Main.java:5)
    at ru.gb.jcore.Main.div1(Main.java:8)
    at ru.gb.jcore.Main.div2(Main.java:11)
    at ru.gb.jcore.Main.main(Main.java:14)
```

Рис. 1.24: Результат запуска цепочки методов (листинг 1.32)

`ArithmeticException` является наследником класса `RuntimeEcxeption` поэтому статический анализатор кода его не подчеркнул, и «ловить» его не обязательно.

При работе с исключениями часто можно встретить слова, похожие на сленг, но это не так. Чаще всего, то, что звучит как сленг – просто перевод ключевых слов языка, осуществляющих то или иное действие.

- try – (англ. пробовать) пробовать, пытаться;
- catch – (англ. ловить) ловить, поймать, хватать;
- throw – (англ. бросать) выбрасывать, бросать, кидать;
- NullPointerException – НПЕ, налпоинтер;
- и другие...

Если посмотреть на метод `div0(int a, int b)` с точки зрения программирования, он написан очень хорошо – алгоритм понятен, метод с единственной ответственностью, однако, из

поставленной перед методом задачи очевидно, что он не может работать при всех возможных входных значениях. То есть если вторая переменная равна нулю, то это ошибка. Необходимо запретить пользователю передавать в качестве делителя ноль. Самое простое – ничего не делать, но в программе на языке Java так нельзя, если мы объявили, что метод имеет возвращающее значение, он обязан что-то вернуть.

Листинг 1.33: Ошибка – нельзя ничего не возвращать

```
private static int div0(int a, int b) {
   if (b != 0)
      return a / b;
   return ???; // ошибка
}
```

А что вернуть, неизвестно, ведь от метода ожидается результат деления. Поэтому, возможно руками сделать проверку (`b == 0f`) и «выбросить» пользователю так называемый **объект исключения** с текстом ошибки, объясняющим произошедшее, а иначе вернём `a / b`.

Листинг 1.34: Цепочка методов

```
private static int div0(int a, int b) {
   if (b == 0f)
      throw new RuntimeException("parameter error");
   return a / b;
}
```

Следовательно, если делитель не равен нулю произойдёт обычное деление, а если равен – будет «выброшено» исключение.

```
Exception in thread "main" java.lang.RuntimeException Create breakpoint : parameter error
    at ru.gb.jcore.Main.div0(Main.java:5)
    at ru.gb.jcore.Main.div1(Main.java:9)
    at ru.gb.jcore.Main.div2(Main.java:12)
    at ru.gb.jcore.Main.main(Main.java:15)
```

Рис. 1.25: Исключение, выброшенное «на наших условиях» (листинг 1.34)

Очевидно, что ключевое слово `new` вызывает конструктор, нового объекта какого-то класса, в который передаётся какой-то параметр, в данном конкретном случае это строка с сообщением.

Объект исключения

Ключевое слово `throw` заставляет созданный объект исключения начать свой путь по родительским методам, пока этот объект не встретится с каким-то обработчиком. В данном конкретном случае – это обработчик виртуальной машины (по-умолчанию), который в специальный поток error выводит так называемый stacktrace, и завершает дальнейшее выполнение метода (технически, всего потока целиком).

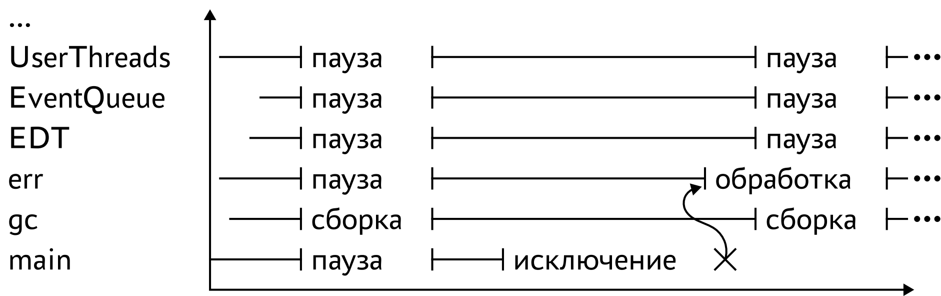


Рис. 1.26: Принципиальная схема работы приложения

Все программы в Java всегда многопоточны. На старте программы запускаются так называемые потоки, которые работают псевдопараллельно и предназначены каждый для решения своих собственных задач, например, это основной поток, поток сборки мусора, поток обработчика ошибок, потоки графического интерфейса. Основная задача этих потоков – делать своё дело и иногда обмениваться информацией.

В **stacktrace**, содержащийся в объекте исключения, кладётся максимальное количество информации о типе исключения, его сообщении, иерархии методов, вызовы которых привели к исключительной ситуации.

Важно научиться читать stacktrace на как можно более раннем этапе изучения программирования.

Итак стектрейс. В стектрейсе на рис. 1.25 видно, что исключение было создано в потоке main, и является объектом класса `RuntimeException`, сообщение также было предусмотрительно приложено автором кода. Важно понять, что исключение – это объект класса. Далее, можно просто читать последовательно строку за строкой – в каком методе создался этот объект, на какой строке, в каком классе. Далее видно, какой код вызвал этот метод, на какой строке, в каком классе.

Если не написать явного выбрасывания никакого исключения, оно всё равно будет выброшено. Это общее поведение исключения. Оно где-то случается, прекращает выполнение текущего метода, и начинает лететь по стеку вызовов вверх. Возможно даже долетит до обработчика по-умолчанию. Некоторые исключения создаются в коде явно, некоторые самой Java, они вполне стандартные, например выход за пределы массива, деление на ноль, и классический `NullPointer`.

Создадим экземпляр класса исключения внутри метода, вызываемого из `main`:

Листинг 1.35: Инициализация объекта исключения

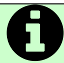
```
1 RuntimeException e = new RuntimeException();
```

Если оставить программу в таком виде и запустить, то ничего не произойдёт, исключение нужно выкинуть (активировать, возбудить, сгенерировать). Для этого есть ключевое слово

Листинг 1.36: Выбрасывание объекта исключения

```
1 throw e;
```

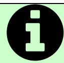
Компилятор ошибок не обнаружил и всё пропустил, а интерпретатор наткнулся на класс исключения, и написал в консоль, что в основном потоке программы возникло исключение в пакете в классе на такой-то строке. По стэктрейсу возможно проследить что откуда вызвалось и как программа дошла до исключительной ситуации. Возможно также наследоваться от какого-то исключения и создать свой класс исключений.

 Исключения, наследники `RuntimeException`, являются **Unchecked**, то есть не обязательные для обработки на этапе написания кода. *Все остальные* `Throwable` – обязательные для обработки, статический анализатор кода не просто их выделяет, а обязывает их обрабатывать на этапе написания кода. И просто не скомпилирует проект если в коде есть необработанные исключения, также известные как **Checked**.

Обработка

Первое, и самое важное, что нужно понять – почему что-то пошло (или пойдёт) «не так», поэтому не пытайтесь что-то ловить, пока не поймёте что именно произошло, от этого понимания будет зависеть *способ* ловли.

Исключение ловится двухсекционным оператором **`try...catch`**, а именно, его первой секцией `try`. Это секция, в которой предполагается возникновение исключения, и предполагается, что его возможно обработать. А в секции `catch` пишется имя класса исключения, которое будет поймано ловим, и имя объекта (идентификатор), через который внутри секции можно к пойманному объекту обращаться. Секция `catch` ловит указанное исключение и *всех его наследников*.

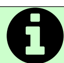
 Рекомендуется писать максимально узко направленные секции `catch`, потому что надо стараться досконально знать как работает программа, и какие исключения она может выбрасывать. Также, потому что разные исключения могут по-разному обрабатываться.

Секций `catch` может быть любое количество. Как только объект исключения обработан, он уничтожается и в следующие `catch` не попадает. Однако, объект возможно явно отправить на обработчик «выше», ключевым словом `throw` (чаще всего, используется `RuntimeException` с конструктором копирования).

Когда какой-то метод выбрасывает исключение у разработчика есть два основных пути:

- обязанность вынести объявление этого исключения в сигнатуру метода, что будет говорить тем, кто его вызывает о том, что в методе может возникнуть исключение;
- исключение необходимо непосредственно в методе обработать, иначе ничего не скомпилируется.

В случае, если объявление исключения выносится в сигнатуру, вызывающий метод должен обработать это исключение точно таким-же образом – либо в вызове, либо вынести в сигнатуру. Исключением из этого правила является класс `RuntimeException`. Все его наследники, включая его самого, обрабатывать не обязательно. Обычно, уже по названию понятно что случилось, и, помимо говорящих названий, там содержится много информации, например, номер строки, вызвавшей исключительную ситуацию.

 Общее правило работы с исключениями одно – если исключение штатное – его надо сразу обработать, если нет – надо дождаться, пока программа упадёт.

Общий вид оператора try...catch можно описать следующим образом:

```
try {
    метод, выбрасывающий исключение
} catch (имя класса исключения и идентификатор) {
    команды, обрабатывающие исключение
}
```

Если произошло исключение, объект исключения попадает в `catch`, и управление ходом выполнения программы попадает в эту секцию. Чаще всего, здесь содержится код, помогающей программе не завершиться. Очень часто в процессе разработки нужно сделать так, чтобы в процессе выполнения что-то конкретное об исключении выводилось на экран, для этого у экземпляра есть метод `getMessage()`.

Листинг 1.37: Получение сообщения из объекта исключения

```
System.out.println(e.getMessage());
```

Ещё чаще бывает, что выполнение программы после выбрасывания исключения не имеет смысла и нужно, чтобы программа завершилась. В этом случае принято выбрасывать новое RuntimeException, передав в него экземпляр пойманного исключения, используя конструктор копирования.

Листинг 1.38: «Проброс» исключения на основе пойманного

```
try {
    // ...
} catch(Exception e) {
    throw new runtimeexception(e);
}
```

Второй вариант обработки исключений – в сигнатуре метода пишется

Листинг 1.39: Обработка исключений в сигнатуре

```
throws IOException,
```

и, через запятую, все остальные возможные исключения этого метода. После этого, с ним не будет проблем исполнения, но у метода который его вызовет – появилась необходимость обработать все checked исключения вызываемого. И так далее наверх.

Задание для самопроверки

1. Исключение – это
   (a) событие в потоке исполнения;
   (b) объект, передаваемый от потока обработчику;
   (c) и то и другое верно.
2. Обработчик исключений – это объект, работающий
   (a) в специальном ресурсе
   (b) в специальном потоке
   (c) в специальный момент
3. Стектрейс - это
   (a) часть потока выполнения программы;
   (b) часть объекта исключения;
   (c) часть информации в окне отладчика.

Пример

Для примера обработки исключений, возникающих на разных этапах работы приложения (жизненного цикла объекта) предлагается описать класс (листинг 1.40), бизнес логика которого подразумевает создание, чтение некоторой информации, например, как если бы нужно было прочитать байт из файла, и закрытие потока чтения, то есть возврат файла обратно под управление ОС.

Листинг 1.40: Экспериментальный класс

```
public class TestStream {
  TestStream() {
    System.out.println("constructor");
  }
  int read() {
    System.out.println("read");
    return 1;
  }
  public void close() {
    System.out.println("close");
  }
}
```

То есть, способ работы с объектом данного класса (полностью без ошибок и других нештатных ситуаций) будет иметь следующий вид

Листинг 1.41: Работа в штатном режиме

```
TestStream stream = new TestStream();
int a = stream.read()
stream.close()
```

Для примера, внутри метода чтения создаётся `FileInputStream` который может генерировать обязательный к проверке на этапе написания кода `FileNotFoundException`, который является наследником `IOException`, который, в свою очередь, наследуется от `Exception`.

Возникает два варианта: либо обернуть в `try...catch`, либо совершенно непонятно, как должна обрабатываться данная исключительная ситуация, и обработать её должна сторона, которая вызывает метод чтения, в таком случае пишется, что метод может выбрасывать исключения. И тогда `TestStream` компилируется без проблем, а вот `main` скомпилироваться уже не может. В нём нужно оборачивать в `try...catch`.

Листинг 1.42: Метод чтения

```
  int read() {
    FileInputStream s = new FileInputStream("file.txt");
    System.out.println("read");
    return 1;
  }
}
```

Листинг 1.43: Обработка исключения

```
try {
   TestStream stream = new TestStream();
   int a = stream.read()
   stream.close()
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
```

Важный момент. Задачи бывают разные. Исключения – это инструмент, который нетривиально работает. Важно при написании кода понять, возникающая исключительная ситуация – штатная, или нештатная. В большинстве случаев – ситуации нештатные, поэтому надо «уронить» приложение и разбираться с тем, что именно произошло. Допустим, для вашего приложения обязательно какой-то файл должен быть, без него дальше нет смысла продолжать. Что делать, если его нет? Ситуация явно нештатная. Самое плохое, что можно сделать – ничего не делать. Это самое страшное, когда программа повела себя как-то не так, а ни мы, разработчики, ни пользователь об этом даже не узнали. Допустим, мы хотим прочитать файл, вывести в консоль, но мы в обработчике исключения просто выводим стектрейс куда-то, какому-то разработчику в среду разработки, и наши действительно важные действия не выполнились. Надо завершать работу приложения. Как завершать? `throw new RuntimeException(e)`. Крайне редко случаются ситуации, когда у исключения достаточно распечатать стектрейс.

Потоки ввода-вывода всегда нужно закрывать. Предположим, что в тестовом потоке открылся файл, из него что-то прочитано, потом метод завершился с исключением, а файл остался незакрытым, ресурсы заняты. Дописав класс `TestStream` при работе с ним, возвращаем из `read` единицу и логируем, что всё прочитали в `main`.

Листинг 1.44: Экспериментальный класс

```
public class TestStream {
  TestStream() {
    System.out.println("constructor");
  }
  int read() throws IOException {
    throw new IOException("read except");
    System.out.println("read");
    return 1;
  }
  public void close() {
    System.out.println("close");
  }
}
```

Далее представим, что в методе `read` что-то пошло не так, выбрасываем исключение, и видим в консоли, что поток создан, произошло исключение, конец программы. Очевидно, поток не закрылся. Что делать?

Делать секцию `finally`. Секция `finally` будет выполнена в любом случае, не важно, будет ли поймано секциями `catch` какое-то исключение, или нет. Возникает небольшая проблема видимости, объявление идентификатора тестового потока необходимо вынести за пределы секции `try`.

Теперь немного неприятностей. Написанный блок `finally`, вроде решает проблему с закрытием потока. А как быть, если исключение возникло при создании этого потока, в конструкторе?

Листинг 1.45: Проблема в конструкторе

```
public class TestStream {
  TestStream() throws IOException {
    throw new IOException("construct except");
    System.out.println("constructor");
  }
  int read() throws IOException {
    throw new IOException("read except");
    System.out.println("read");
    return 1;
  }
  public void close() {
    System.out.println("close");
  }
}
```

Метод закрытия будет пытаться выполниться от ссылки на `null`. Недопустимо.

При возникновении в конструкторе потока `IOException` - получим `NullPointerException` в блоке `finally`.

Очевидное решение – поставить в секции `finally` условие, и если поток не равен `null`, закрывать. Это точно сработает. Меняем тактику.

Листинг 1.46: Проблема при закрытии

```
public class TestStream {
  TestStream() throws IOException {
    throw new IOException("construct except");
    System.out.println("constructor");
  }
  int read() throws IOException {
    throw new IOException("read except");
    System.out.println("read");
    return 1;
  }
  public void close() throws IOException {
    throw new IOException("close except");
    System.out.println("close");
  }
}
```

Конструктор отрабатывает нормально. Метод чтения всё ещё генерирует исключение, но и в методе закрытия что-то пошло не так, и вылетело исключение. Нужно оборачивать в `try...catch`. Итоговый код, работающий с классом будет иметь следующий вид.

Листинг 1.47: Обработка исключений, насколько это возможно

```
TestStream stream = null;
try {
   stream = new TestStream();
   int a = stream.read()
   stream.close()
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
} finally {
   try {
      stream.close();
   } catch (NullPointerException e) {
      e.printStackTrace();
   }
}
```

Но и тут возможно наткнуться на неприятность. Допустим, что необходимо в любом случае ронять приложение.

Листинг 1.48: Проблемы подавленных исключений

```
 // ...
catch (IOException e) {
   throw new RuntimeException(e);
} finally {
   try {
      stream.close();
   } catch (NullPointerException e) {
      e.printStackTrace();
   }
}
```

Тогда если `try` поймал исключение, и выкинул его, потом `finally` всё равно выполнится, и второе исключение перекроет (подавит) первое, никто его не увидит. Хотя по логике, первое для работы важнее. Так было до Java 1.8.

try-with-resources block

Начиная с версии Java 1.8 разработчику предоставляется механизм **try-с-ресурсами**. Поток – это ресурс, абстрактное понятие. Выражаясь строго формально, разработчик должен *реализовать интерфейс* `Closeable`. В этом интерфейсе содержится всего один метод `close()`, который умеет бросать `IOException`. В классе тестового потока нужно всего лишь переопределить этот метод данного интерфейса.

Листинг 1.49: Реализация интерфейса закрытия потока

```
public class TestStream implements Closeable {
   // ...

   @Override
```

```
    public void close() throws IOException {
        throw new IOException("close except");
        System.out.println("close");
    }
}
```

Все потоки начиная с Java 1.8 реализуют интерфейс `Closeable`. Работа с такими классами имеет лаконичный вид

Листинг 1.50: Реализация блока try-with-resources

```
try (TestStream stream = new TestStream()) {
    int a = stream.read();
} catch (IOException e) {
    throw new RuntimeException(e)
}
```

В данном коде не нужно закрывать поток явно, это будет сделано автоматически в следствие реализации интерфейса. Если ломается метод `read()`, то try-с-ресурсами всё равно корректно закроет поток. При сломанном методе закрытия и сломанном методе чтения одновременно, JVM запишет наверх основное исключение, но и выведет «подавленное» исключение, вторичное в стектрейс. Рекомендуется по возможности всегда использовать try-с-ресурсами.

Наследование и полиморфизм исключений

Наследование и полиморфизм для исключений – тема не очень большая и не сложная, потому что ничего нового в информацию про классы и объекты исключения не привносят. Застрять внимание на объектно-ориентированном программировании в исключениях не целесообразно, потому что исключения это *тоже классы* и те исключения, которые используются в программе – уже какие-то наследники других исключений.

Генерируются и выбрасываются *объекты исключений*, единственное, что важно упомянуть это то, что подсистема исключений работает не тривиально. Но разработчик может создавать собственные исключения с собственными смыслами и сообщениями и точно также их выбрасывать вместо стандартных. Наследоваться возможно от любых исключений, единственное что важно, это то, что не рекомендуется наследоваться от классов `Throwable` и `Error`, когда описываете исключение.

Механика checked и unchecked исключений сохраняется при наследовании, поэтому создав наследник `RuntimeException` вы получаете не проверяемые на этапе написания кода исключения.

Практическое задание

1. напишите два наследника класса Exception: ошибка преобразования строки и ошибка преобразования столбца
2. разработайте исключения-наследники так, чтобы они информировали пользователя в формате ожидание/реальность
3. для проверки напишите программу, преобразующую квадратный массив целых чисел 5x5 в сумму чисел в этом массиве, при этом, программа должна выбросить исключение, если строк или столбцов в исходном массиве окажется не 5.

Термины, определения и сокращения

`finally` — часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.

`throws` — ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.

`throw` — оператор, активирующий (выбрасывающий) объект исключения.

`try...catch` — двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.

BIOS — (англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.

CI — (англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CLI — (англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».

cp1251 — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

1.5. Специализация: тонкости работы

В предыдущем разделе

Рассмотрены понятия внутренних и вложенных классов; процессы создания, использования и расширения перечислений. Подробно рассмотрены исключения с точки зрения ООП, их философия и тесная связь с многопоточностью в Java, обработка, разделение понятия штатных и нештатных ситуаций.

В этом разделе

Файловые системы и представление данных в запоминающих устройствах; Начало рассмотрения популярных пакетов ввода-вывода `java.io`, `java.nio`. Более подробно будет разобран один из самых популярных ссылочных типов данных `String` и механики вокруг него.

- Файл;
- Загрузчик;
- Разделы;
- BIOS;
- ФС;
- FAT;
- NTFS;
- Ввод-вывод;
- Потоки в-в;
- Строка;
- String;

1.5.1. Файловая система

Повествование вплотную подошло к работе языка в части общения программы со внешним миром, а делать это не разобравшись в тонкостях работы файловой системы не эффективно. В этом подразделе находится материал, напрямую не относящийся к Java.

Файловая система (File System) – FS, ФС – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

MBR, GPT

ОС Linux, например, предоставляет возможность разбивки жесткого диска компьютера на отдельные разделы. Пользователи могут определить их границы по так называемым таблицам разделов. **Основная загрузочная запись (MBR)** и **таблица разделов GUID (GPT)** – это два стиля формата разделов, которые позволяют компьютеру загружать операционную систему с жесткого диска, а также индексировать и упорядочивать данные.

Основная загрузочная запись (MBR) – устаревшая форма разделения загрузочного сектора – первый сектор диска, который содержит информацию о том, как разбит диск. Он также содержит загрузчик, который сообщает компьютеру, как загрузить ОС. Main Boot Record состоит из трех частей:

– Основной загрузчик – MBR резервирует первые байты дискового пространства для основного загрузчика. Windows размещает здесь очень упрощенный загрузчик, в то время как другие ОС могут размещать более сложные многоступенчатые загрузчики.
– Таблица разделов диска – таблица разделов диска находится в нулевом цилиндре, нулевой головке и первом секторе жёсткого диска. Она хранит информацию о том, как разбит диск. MBR выделяет 16 байт данных для каждой записи раздела и может выделить всего 64 байта. Таким образом, Main Boot Record может адресовать не более четырёх основных разделов или трёх основных раздела и один расширенный раздел.
– Конечная подпись – это 2-байтовая подпись, которая отмечает конец MBR. Всегда устанавливается в шестнадцатеричное значение `0x55AA`.

Вообще, программисты любят всякие шестнадцатеричные подписи вроде `0xDEADBEEF`, `0xA11F0DAD`, `0xDEADFA11`, одну из таких подписей можно было увидеть в первом разделе, `0xcafebabe`. Они называются hexspeak.

Таблица разделов GUID (GPT) – это стиль формата раздела, который был представлен в рамках инициативы United Extensible Firmware Interface (UEFI). GPT был разработан для архитектурного решения некоторых ограничений MBR. Стиль GPT новее, гибче и надежнее, чем MBR. GPT использует логическую адресацию блоков для указания блоков данных. Первый блок помечен как LBA0, затем LBA1, LBA2, и так далее GPT хранит защитную запись MBR в LBA0, свой основной заголовок в логическом блоке и записи разделов в блоках со второго по тридцать третий.

GPT:

– Защитная MBR;
– Основной тег GPT;
– Записи разделов;
– Дополнительный GPT.
– Максимальная емкость раздела 9.4ZB;
– 128 первичных разделов;
– Устойчив к повреждению первичного раздела, так как имеет вторичный GPT;
– Возможность использовать функции UEFI.

Файловая система – это архитектура хранения информации, размещенной на жестком диске и в оперативной памяти. С её помощью пользователь получает доступ к структуре ядра системы.

На что влияет файловая система?

1. скорость обработки данных;
2. сбережение файлов;
3. оперативность записи;
4. допустимый размер блока;
5. возможность хранения информации в оперативной памяти;
6. способы корректировки пользователями ядра
7. и пр.

Linux

ОС Linux предоставляет возможность устанавливать в каждый отдельный блок свою файловую систему, которая и будет обеспечивать порядок поступающих и хранящихся данных,

поможет с их организацией. Каждая ФС работает на наборе правил. Исходя из этого и определяется в каком месте и каким образом будет выполняться хранение информации. Эти правила лежат в основе иерархии системы, то есть всего корневого каталога. От того, насколько правильно администратор компьютера выберет тип файловой системы для каждого раздела, зависит ряд параметров, таких как: оперативность записи, скорость обработки данных, сбережение файлов, допустимый размер блока, возможность хранения информации в оперативной памяти и другие.

В файловой системе, хранятся файлы. Файлы в Linux – это особенная отдельная категория данных. С точки зрения ОС – всё файл. Все файлы делятся на категории.

- Regular File – Предназначены для хранения двоичной и символьной информации. Это жесткая ссылка, ведущая на фактическую информацию, размещенную в каталоге. Если этой ссылке присвоить уникальное имя, получим Named Pipe, то есть именованный канал.
- Device File – файлы для устройств, туннелей. Речь идет о физических устройствах, представленных в ОС файлами. Могут классифицировать специальные символы и блоки. Обеспечивают мгновенный доступ к дисководам, принтерам, модемам, воспринимая их как простой файл с данными.
- Soft Link – мягкая (символьная) ссылка. Отвечает за мгновенный доступ к файлам, размещенным на любых носителях информации. В процессе копирования, перемещения и других действий пользователя, работающего по ссылке, будет выполняться операция над документом, на который ссылаются.
- Directories – Каталоги. Обеспечивают быстрый и удобный доступ к каталогам. Представляет собой файл с директориями и указателями на них. Это некого рода картотека: в папках размещаются документы, а в директориях – дополнительные каталоги.
- Block devices and sockets – Блочные и символьные устройства. Выделяют интерфейс, необходимый для взаимодействия приложений с аппаратной составляющей. Каналы и сокеты. Отвечают за взаимодействие внутренних процессов в операционной системе.

Windows

Линейка файловых систем для Windows – роль в работе системы и этапы развития.

FAT(16) File Allocation Table Использовалась для MS-DOS 3.0, Windows 3.x, Windows 95, Windows 98, Windows NT/2000. Была разработана достаточно давно и предназначалась для работы с небольшими дисковыми и файловыми объемами, простой структурой каталогов. Таблица размещается в начале тома, причем хранятся две ее копии (в целях обеспечения большей устойчивости). Данная таблица используется операционной системой для поиска файла и определения его физического расположения на жестком диске. В случае повреждения и таблицы и ее копии чтение файлов операционной системой становится невозможно.

Файловая система FAT32

- возможность перемещения корневого каталога
- возможность хранения резервных копий

Начиная с Windows 95, компания Microsoft начинает активно использовать в своих операционных системах FAT32 – тридцатидвухразрядную версию FAT. FAT32 стала обеспечивать более оптимальный доступ к дискам, более высокую скорость выполнения операций ввода/вывода, а также поддержку больших файловых объемов (объем диска до 2 Тбайт).

Файловая система NTFS – (от англ. New Technology File System), файловая система новой технологии

- права доступа
- Шифрование данных
- Дисковые квоты
- Хранение разреженных файлов

– Журналирование

Ни одна из версий FAT не обеспечивает хоть сколько-нибудь приемлемого уровня безопасности. Это, а также необходимость в добавочных файловых механизмах (сжатия, шифрования) привело к необходимости создания принципиально новой файловой системы. NTFS. В ней для файлов и папок могут быть назначены права доступа (на чтение, на запись и т.д.). Благодаря этому существенно повысилась безопасность данных и устойчивость работы системы. Одной из основных целей создания NTFS было обеспечение скоростного выполнения операций над файлами (копирование, чтение, удаление, запись), а также предоставление дополнительных возможностей: сжатие данных, восстановление поврежденных файлов системы на больших дисках и т.д. Другой основной целью создания NTFS была реализация повышенных требований безопасности.

Файловая система FAT для современных жестких дисков просто не подходит (ввиду ее ограниченных возможностей). Что касается FAT32, то ее еще можно использовать, но уже с оговорками.

Файловые системы

– Ext (extended) FS. Это расширенная файловая система, одна из первых. Была запущена в работу еще в 1992 году. В основе ее функциональности лежала ФС UNIX. Основная задача состояла в выходе за рамки конфигурации классической файловой системы MINIX, исключить ее ограничения и повысить эффективность администрирования. Сегодня она применяется крайне редко.
– Ext2. Вторая, более расширенная вервия ФС, паявшаяся на рынке в 1993 году. По своей структуре продукт аналогичный Ext. Изменения коснулись интерфейса, конфигурации. Увеличился объем памяти, производительность. Максимально допустимый объем файлов для хранения (указывается в настройках) – 2 ТБ. Ввиду невысокой перспективности применяется на практике редко.
– Ext3. Третье поколение Extended FS, введенное в использование в 2001 году. Уже относится к журналируемой. Позволяет хранить логи – изменения, обновления файлов данных записываются в отдельный журнал еще до того, как эти действия будут завершены. После перезагрузки ПК, такая ФС позволит восстановить файлы благодаря внедрению в систему специального алгоритма.
– Ext4. Четвертое поколение Extended FS, запущенное в 2006 году. Здесь максимально убраны всевозможные ограничения, присутствующие в предыдущих версиях. Сегодня именно она по умолчанию входит в состав большей части дистрибутивов Линукс. Передовой ее нельзя назвать, но стабильность и надежность работы здесь в приоритете. В Unix системах применяется повсеместно.
– JFS. Журналируемая система, первый аналог продуктам из основной группы, разработанная специалистами IBM под AIX UNIX. Отличается постоянством и незначительными требованиями к работе. Может использоваться на многопроцессорных ПК. Но в ее журнале сохраняются ссылки лишь на метаданные. Поэтому если произойдет сбой, автоматически подтянутся устаревшие версии данных.
– ReiserFS. Создана Гансом Райзером исключительно под Linux и названа в его честь. По своей структуре – это продукт, похожий на Ext3, но с более расширенными возможностями. Пользователи могут соединять небольшие файлы в более масштабные блоки, исключая фрагментацию, повышая эффективность функционирования. Но в случае непредвиденного отключения электроэнергии есть вероятность потерять данные, которые будут группироваться в этот момент.
– XFS. Еще один представитель группы журналируемых файловых систем. Отличительная особенность: в логи программа будет записывать только изменения в метаданных. Из пре-

имуществ выделяют быстроту работы с объемной информацией, способность выделять место для хранения в отложенном режиме. Позволяет увеличивать размеры разделов, а вот уменьшать, удалять часть – нельзя. Здесь также есть риск потери данных при отключении электроэнергии.

– Btrfs. Отличается повышенной стойкостью к отказам и высокой производительностью. Удобная в работе, позволяет легко восстанавливать информацию, делать скриншоты. Размеры разделов можно менять в рабочем процессе. По умолчанию входит в OpenSUSE и SUSE Linux. Но обратная совместимость в ней нарушена, что усложняет поддержку.
– F2FS. Разработка Samsung. Уже входит в ядро Linux. Предназначена для взаимодействия с хранилищем данных флеш-памяти. Имеет особую структуру: носитель разбивается на отдельные части, которые в свою очередь дополнительно еще делятся.
– OpenZFS. Это ответ на вопрос какую файловую систему выбрать для Ubuntu – она автоматически включена в поддержку ОС уже более 6 лет назад. Отличается высоким уровнем защиты от повреждения информации, автоматическим восстановлением, поддержкой больших объемов данных.
– EncFS. Шифрует данные и пересохраняет их в этом формате в указанную пользователем директорию. Надо примонтировать ФС чтобы обеспечить доступ в расшифрованной информации.
– Aufs. С ее помощью отдельные File Systems можно группировать в один раздел.
– NFS. Позволит через сеть примонтировать ФС удаленного устройства.
– Tmpfs. Предусмотрена возможность размещения пользовательских файлов непосредственно в оперативной памяти ПК. Предполагает создание блочного узла определенного размера с последующим подключением к папке. При необходимости данные можно будет удалять.
– Procfs. По умолчанию размещена в папке proc. Буде содержать полный набор данных относительно процессов, запущенных в системе и непосредственно в ядре в режиме реального времени.
– Sysfs. Такая ФС позволит пользователю задавать и отменять параметры ядра во время выполнения задачи.

Задание для самопроверки

1. MBR – это
   (a) main boot record;
   (b) master BIOS recovery;
   (c) minimizing binary risks.
2. Что такое GPT?
   (a) General partition trace;
   (b) GUID partition table;
   (c) Greater pass timing.
3. Открытый вопрос: Что такое файловая система?
4. Возможно ли использовать разные файловые системы в рамках одной ОС?

1.5.2. Файловая система и представление данных

File

До появления Java 1.7 все операции проводились с помощью класса `File`.
В Java есть специальный класс (File), с помощью которого можно управлять файлами

на диске компьютера. Для того чтобы управлять содержимым файлов, есть другие классы: `FileInputStream`, `FileOutputStream` и другие. Под управлением файлами понимается, что их можно создавать, удалять, переименовывать, узнавать их свойства и пр.

Листинг 1.51: Создание объекта файла

```java
File file = new File("file.txt");
```

Здесь происходит привычное создание объекта, причём в параметре конструктора задано так называемое относительное имя файла, то есть только имя и расширение, без указания полного пути, а значит файл будет открыт там, где исполняется программа.

С помощью объекта файла возможно работать и с директориями.

Листинг 1.52: Использования файла для директории

```java
File folder = new File(".");
for (File file : folder.listFiles())
   System.out.println(file.getName());
```

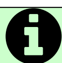
Такой код выведет на экран всё содержимое *текущей* директории, о чём говорит точка в строке с именем файла. Если просто указать имя файла, то будет использован файл в той же папке, что исполняемая программа, а если указана точка, то это означает использование непосредственно данной папки. метод `listFiles()` возвращает список файлов из указанной папки. А метод `getName()` выдаёт имя файла с расширением.

Листинг 1.53: Методы класса `File`

```java
System.out.println("Is it a folder - " + folder.isDirectory());
System.out.println("Is it a file - " + folder.isFile());
File file = new File("./Dockerfile");
System.out.println("Length file - " + file.length());
System.out.println("Absolute path - " + file.getAbsolutePath());
System.out.println("Total space on disk - " + folder.getTotalSpace());
System.out.println("File deleted - " + file.delete());
System.out.println("File exists - " + file.exists());
System.out.println("Free space on disk - " + folder.getFreeSpace());
```

Класс файл предоставляет ряд методов для манипуляции файлами и директориями.

- проверить, является ли объект файлом или директорией;
- узнать размер файла в байтах и его абсолютный путь;
- порядковый номер жёсткого диска, на котором расположен файл;
- удалить файл (метод возвращает истину или ложь, как результат);
- проверить, существует ли такой файл;
- узнать, сколько ещё свободного места доступно на диске;
- и другие.



Фактически у класса `File` почти все методы дублированы: одна версия возвращает (и принимает в качестве параметра) `String`, вторая `File`.

Paths, Path, Files, FileSystem

В Java 1.7 создатели языка решили изменить работу с файлами и каталогами.

У класса File существует ряд недостатков. Например, в нем нет метода copy(), который позволил бы скопировать файл. В классе File достаточно много методов, которые возвращают boolean значения.

Вместо единого класса File появились три класса: Paths, Path и Files. Также появился класс FileSystem, который предоставляет интерфейс к файловой системе.

URI – Uniform Resource Identifier (унифицированный идентификатор ресурса);
URL – Uniform Resource Locator (унифицированный определитель местонахождения ресурса);
URN – Unifrorm Resource Name (унифицированное имя ресурса).

Paths – это совсем простой класс с единственным статическим методом get(). Его создали исключительно для того, чтобы из переданной строки или URI получить объект типа Path.

Фактически, Path – это переработанный аналог класса File. Работать с ним значительно проще, чем с File. Например, метод getParent(), возвращает родительский путь для текущего файла в виде строки. Но при этом есть метод getParentFile(), который возвращал то же самое, но в виде объекта File. Это явно избыточно.

Листинг 1.54: Использование классов Path и Paths

```
Path filePath = Paths.get("pics/logo.png");

Path fileName = filePath.getFileName();
System.out.println("Filename: " + fileName);
Path parent = filePath.getParent();
System.out.println("Parent directory: " + parent);

boolean endWithTxt = filePath.endsWith("logo.png");
System.out.println("Ends with filepath: " + endWithTxt);
endWithTxt = filePath.endsWith("png");
System.out.println("Ends with string: " + endWithTxt);

boolean startsWithPics = filePath.startsWith("pics");
System.out.println("Starts with filepath: " + startsWithPics);
```

В методы startsWith() и endsWith() нужно передавать путь, а не просто набор символов: в противном случае результатом всегда будет false, даже если текущий путь действительно заканчивается такой последовательностью символов.

```
Filename: logo.png
Parent directory: pics
Ends with filepath: true
Ends with string: false
Starts with filepath: true
```

Метод normalize() – «нормализует» текущий путь, удаляя из него ненужные элементы.

При обозначении путей часто используются символы "." (для обозначения текущей директории) и ".." (для родительской директории).

Если в программе появился путь, использующий "." или "..", метод `normalize()` позволит удалить их и получить путь, в котором они не будут содержаться.

Листинг 1.55: Метод `normailze()`

```
Path path = Paths.get("./sources-draft/../pics/logo.png");
System.out.println(path.normalize());
```

```
pics/logo.png
```

`Files` — это утилитарный класс, куда были вынесены статические методы из класса `File`. Он сосредоточен на управлении файлами и директориями.

Листинг 1.56: Использование класса `Files`

```
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

Path file = Files.createFile(Paths.get("../pics/file.txt"));
System.out.print("Was the file captured successfully in pics directory? ");
System.out.println(Files.exists(Paths.get("../pics/file.txt")));

Path testDirectory = Files.createDirectory(Paths.get("./testing"));
System.out.print("Was the test directory created successfully? ");
System.out.println(Files.exists(Paths.get("./testing")));

file = Files.move(file, Paths.get("./testing/file.txt"), REPLACE_EXISTING);
System.out.print("Is our file still in the pics directory? ");
System.out.println(Files.exists(Paths.get("../pics/file.txt")));
System.out.print("Has our file been moved to testDirectory? ");
System.out.println(Files.exists(Paths.get("./testing/file.txt")));

Path copyFile = Files.copy(file, Paths.get("../pics/file.txt"),
    REPLACE_EXISTING);
System.out.print("Has our file been copied to pics directory? ");
System.out.println(Files.exists(Paths.get("../pics/file.txt")));

Files.delete(file);
System.out.print("Does the file exist in test directory? ");
System.out.println(Files.exists(Paths.get("./testing/file.txt")));
System.out.print("Does the test directory exist? ");
System.out.println(Files.exists(Paths.get("./testing")));
```

```
Was the file captured successfully in pics directory? true
Was the test directory created successfully? true
Is our file still in the pics directory? false
Has our file been moved to testDirectory? true
Has our file been copied to pics directory? true
```

```
Does the file exist in test directory? false
Does the test directory exist? true
```

Класс Files позволяет не только управлять самими файлами, но и работать с его содержимым. Для записи данных в файл у него есть метод `write()`, а для чтения: `read()`, `readAllBytes()` и `readAllLines()`.

Листинг 1.57: Использование класса `Files`

```
List<String> lines = Arrays.asList(
      "The cat wants to play with you",
      "But you don't want to play with it");

Path file = Files.createFile(Paths.get("cat.txt"));

if (Files.exists(file)) {
   Files.write(file, lines, StandardCharsets.UTF_8);
   lines = Files.readAllLines(
   Paths.get("cat.txt"), StandardCharsets.UTF_8);

   for (String s : lines) {
      System.out.println(s);
   }
}
```

Задание для самопроверки

Ссылка на местонахождение – это:

1. URI;
2. URL;
3. URN.

1.5.3. Потоки ввода-вывода, пакет `java.io`

Подавляющее большинство программ обменивается данными со внешним миром. Это делают любые сетевые приложения – они передают и получают информацию от других компьютеров и специальных устройств, подключенных к сети. Можно таким же образом представлять обмен данными между устройствами внутри одной машины. Программа может считывать данные с клавиатуры и записывать их в файл, или, наоборот - считывать данные из файла и выводить их на экран. Таким образом, устройства, откуда может производиться считывание информации, могут быть самыми разнообразными – файл, клавиатура, входящее сетевое соединение и т.д. То же касается и устройств вывода – это может быть файл, экран монитора, принтер, исходящее сетевое соединение и т.п. В конечном счете, все данные в компьютерной системе в процессе обработки передаются от устройств ввода к устройствам вывода.

Реализация системы ввода/вывода осложняется не только широким спектром источников и получателей данных, но еще и различными форматами передачи информации. Ею можно обмениваться в двоичном представлении, символьном или текстовом, с применением некоторой кодировки (кодировок только для русского языка существует более четырёх типов), или передавать числа в различных представлениях. Доступ к данным может потребоваться как после-

довательный, так и произвольный. Зачастую для повышения производительности применяется буферизация.

В Java для описания работы по вводу/выводу используется специальное понятие потока данных (stream). Поток данных это абстракция, физически никакие потоки в компьютере никуда не текут.

Поток связан с некоторым источником, или приемником, данных, способным получать или предоставлять информацию. Потоки делятся на входящие – читающие данные и исходящие – передающие (записывающие) данные. Введение концепции stream позволяет абстрагировать основную логику программы, обменивающейся информацией с любыми устройствами одинаковым образом, от низкоуровневых операций с такими устройствами ввода/вывода. Для программы нет разницы, передавать данные в файл или в сеть, принимать с клавиатуры или со специализированного устройства.

В Java потоки естественным образом представляются объектами. Описывающие их классы составляют основную часть пакета `java.io`. Все классы разделены на две части – одни осуществляют ввод данных, другие – вывод.

Классы `InputStream` и `OutputStream`

Почти все классы пакета, осуществляющие ввод-вывод, так или иначе наследуются от InputStream для входных данных, и для выходных – от OutputStream.

`InputStream` – это базовый абстрактный класс для потоков ввода, т.е. чтения.

`InputStream` описывает базовые методы для работы со входящими байтовыми потоками данных. Простейшая операция представлена методом `read()` (без аргументов). Согласно документации, этот метод предназначен для считывания ровно одного байта из потока, однако возвращает при этом значение типа `int`. В том случае, если считывание произошло успешно, возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт (значение `int` содержит 4 байта и получается простым дополнением нулями в двоичном представлении). Если достигнут конец потока, то есть в нем больше нет информации для чтения, то возвращаемое значение равно -1. Если же считать из потока данные не удается из-за каких-то ошибок, или сбоев, будет брошено исключение `java.io.IOException`. Дело в том, что каналы передачи информации, будь то Internet или, например, жёсткий диск, могут давать сбои независимо от того, насколько хорошо написана программа. А это означает, что нужно быть готовым к ним, чтобы пользователь не потерял нужные данные. Когда работа с входным потоком данных окончена, его следует закрыть. Для этого вызывается метод `close()`. Этим вызовом будут освобождены все системные ресурсы, связанные с потоком.

`OutputStream` – это базовый абстрактный класс для потоков вывода, т.е. записи.

В классе `OutputStream` аналогичным образом определяются три метода `write()` – один принимающий в качестве параметра `int`, второй – `byte[]` и третий – `byte[]`, и два `int`-числа. Все эти методы ничего не возвращают. Для записи в поток сразу некоторого количества байт методу `write()` передается массив байт. Или, если мы хотим записать только часть массива, то передаем массив `byte[]` и два числа – отступ и количество байт для записи. Реализация потока вывода может быть такой, что данные записываются не сразу, а хранятся некоторое время в памяти. Чтобы убедиться, что данные записаны в поток, а не хранятся в буфере, вызывается метод `flush()`. Когда работа с потоком закончена, его следует закрыть, для этого вызывается метод `close()`.

> ⚠ Закрытый поток не может выполнять операции вывода и не может быть открыт заново.

Классы ByteArrayInputStream и ByteArrayOutputStream

Самый естественный для программы и простой для понимания источник, откуда можно считывать байты – это, конечно, массив байт. Класс `ByteArrayInputStream` представляет собой поток, считывающий данные из массива байт. Этот класс имеет конструктор, которому в качестве параметра передается массив `byte[]`. Соответственно, при вызове методов `read()` возвращаемые данные будут браться именно из этого массива. Аналогично, для записи байт в массив применяется класс ByteArrayOutputStream. Этот класс использует внутри себя объект `byte[]`, куда записывает данные, передаваемые при вызове методов `write()`. Чтобы получить записанные в массив данные, вызывается метод `toByteArray()`. В примере будет создан массив, который состоит из трёх элементов: 1, -1 и 0. Затем, при вызове метода `read()` данные считывались из массива, переданного в конструктор `ByteArrayInputStream`. Обратите внимание, в данном примере второе считанное значение равно 255, а не -1, как можно было бы ожидать. Чтобы понять, почему это произошло, нужно вспомнить, что метод `read()` считывает `byte`, но возвращает значение `int`, полученное добавлением необходимого числа нулей (в двоичном представлении).

Листинг 1.58: Использование Byte Array Stream

```
ByteArrayOutputStream out = new ByteArrayOutputStream();

out.write(1);
out.write(-1);
out.write(0);

ByteArrayInputStream in = new ByteArrayInputStream(out.toByteArray());

int value = in.read();
System.out.println("First element is - " + value);

value = in.read();
System.out.println("Second element is - " + value +
      ". If (byte)value - " + (byte)value);

value = in.read();
System.out.println("Third element is - " + value);
```

Классы FileInputStream и FileOutputStream

Класс `FileInputStream` используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то будет брошено

`java.io.FileNotFoundException`. Если же объект успешно создан, то при вызове его методов `read()` возвращаемые значения будут считываться из указанного файла.

Для записи байт в файл используется класс `FileOutputStream`. При создании объектов этого класса, то есть при вызовах его конструкторов, кроме имени файла, также можно указать, будут ли данные дописываться в конец файла, либо файл будет полностью перезаписан.

Если не указан флаг добавления, то всегда сразу после создания `FileOutputStream` файл будет **создан** (содержимое существующего файла будет стёрто).

При вызовах методов `write()` передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод `close()`, чтобы сообщить системе, что работа по записи файла закончена.

Другие потоковые классы

- Классы `PipedInputStream` и `PipedOutputStream` характеризуются тем, что их объекты всегда используются в паре – к одному объекту `PipedInputStream` привязывается (подключается) один объект `PipedOutputStream`. Они могут быть полезны, если в программе необходимо организовать обмен данными между модулями. Более явно выгода от использования проявляется при разработке многопоточных (multithread) приложений.
- `StringBufferInputStream` (deprecated). Иногда бывает удобно работать с текстовой строкой как с потоком байт. Для этого возможно воспользоваться классом `StringBufferInputStream`. При создании объекта этого класса необходимо передать конструктору объект `String`.
- Класс `SequenceInputStream` объединяет поток данных из других двух и более входных потоков. Данные будут вычитываться последовательно – сначала все данные из первого потока в списке, затем из второго, и так далее. Конец потока `SequenceInputStream` будет достигнут только тогда, когда будет достигнут конец потока, последнего в списке.
- FilterInputStream и FilterOutputStream и их наследники. Задачи, возникающие при вводе/-выводе весьма разнообразны – это может быть считывание байтов из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов и т.д. В такой ситуации решение с использованием простого наследования приводит к возникновению слишком большого числа подклассов. Более эффективно применение надстроек (в ООП этот шаблон называется адаптер). Надстройки – наложение дополнительных объектов для получения новых свойств и функций. Таким образом, необходимо создать несколько дополнительных объектов – адаптеров к классам ввода/вывода. В терминах `java.io` их называют фильтрами.
- Класс `LineNumberInputStream` во время чтения данных производит подсчет, сколько строк было считано из потока. Номер строки, на которой в данный момент происходит чтение, можно узнать путем вызова метода `getLineNumber()`. Также можно и перейти к определенной строке вызовом метода `setLineNumber(int lineNumber)`. Этот класс практически сразу объявили устаревшим и вместо него используется `LineNumberReader` с аналогичным функционалом.
- `PushBackInputStream`. Этот фильтр позволяет вернуть во входной поток считанные из него данные. Такое действие производится вызовом метода `unread()`. Понятно, что обеспечивается подобная функциональность за счет наличия в классе специального буфера – массива байт, который хранит считанную информацию.

– PrintStream используется для конвертации и записи строк в байтовый поток. В нем определен метод `print()`, принимающий в качестве аргумента различные примитивные типы Java, а также тип `Object`. При вызове передаваемые данные будут сначала преобразованы в строку, после чего записаны в поток. Если возникает исключение, оно обрабатывается внутри метода `print()` и дальше не бросается (узнать, произошла ли ошибка, можно с помощью метода `checkError()`). Данный класс также считается устаревшим, и вместо него рекомендуется использовать `PrintWriter`, однако старый класс продолжает активно использоваться, поскольку статические поля `out` и `err` класса `System` имеют именно это тип.

BufferedInputStream и BufferedOutputStream

На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. `BufferedInputStream` содержит массив байт, который служит буфером для считываемых данных. То есть, когда байты из потока считываются, либо пропускаются (методом `skip()`), сначала заполняется буферный массив, причём, из потока загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции `read()` или `skip()`. `BufferedOutputStream` предоставляет возможность производить многократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него, произойдет, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод `flush()`. Для наглядности заполним небольшой файл данными, буквально, миллион символов.

Листинг 1.59: Сравнение простого и буферизующего потоков (шаг 1)

```
String fileName = "test.txt";
InputStream inStream = null;
OutputStream outStream = null;
try {
   long timeStart = System.currentTimeMillis();
   outStream = new BufferedOutputStream(new FileOutputStream(fileName));
   for (int i = 1000000; --i >= 0;) { outStream.write(i); }

   long time = System.currentTimeMillis() - timeStart;
   System.out.println("Writing time: " + time + " millisec");
   outStream.close();
} catch (IOException e) {
   System.out.println("IOException: " + e.toString());
   e.printStackTrace();
}
```

```
Writing time: 41 millisec
```

На следующем шаге, прочитав эти символы из файла простым потоком ввода из файла, увидим, что это заняло сколько-то времени.

Листинг 1.60: Сравнение простого и буферизующего потоков (шаг 2)

```
try {
```

```
    long timeStart = System.currentTimeMillis();
    InputStream inStream = new FileInputStream(fileName);
    while (inStream.read() != -1) { }

    long time = System.currentTimeMillis() - timeStart;
    inStream.close();
    System.out.println("Direct read time: " + (time) + " millisec");
} catch (IOException e) {
    System.out.println("IOException: " + e.toString());
    e.printStackTrace();
}
```

```
Direct read time: 2726 millisec
```

На третьем шаге становится очевидно, что буферизующий поток справляется ровно с той-же задачей на пару порядков быстрее. Выгода использования налицо.

Листинг 1.61: Сравнение простого и буферизующего потоков (шаг 3)

```
try {
    long timeStart = System.currentTimeMillis();
    inStream = new BufferedInputStream(new FileInputStream(fileName));
    while (inStream.read() != -1) { }

    long time = System.currentTimeMillis() - timeStart;
    inStream.close();
    System.out.println("Buffered read time: " + (time) + " millisec");
} catch (IOException e) {
    System.out.println("IOException: " + e.toString());
    e.printStackTrace();
}
```

```
Buffered read time: 23 millisec
```

Усложнение данных

До сих пор речь шла только о считывании и записи в поток данных в виде `byte`. Для работы с другими типами данных Java определены интерфейсы `DataInput` и `DataOutput` и их реализации – классы-фильтры `DataInputStream` и `DataOutputStream`, реализующие методы считывания и записи значений всех примитивных типов. При этом происходит конвертация этих данных в набор `byte` и обратно. Чтение необходимо организовать так, чтобы данные запрашивались в виде тех же типов, в той же последовательности, как и производилась запись. Если записать, например, `int` и `long`, а потом считывать их как `short`, чтение будет выполнено корректно, без исключительных ситуаций, но числа будут получены совсем другие.

Листинг 1.62: Использование Data Stream (шаг 1)

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
```

```
import java.io.DataOutputStream;
import java.io.InputStream;

ByteArrayOutputStream out = new ByteArrayOutputStream();
try {
   DataOutputStream outData = new DataOutputStream(out);

   outData.writeByte(128);
   outData.writeInt(128);
   outData.writeLong(128);
   outData.writeDouble(128);
   outData.close();
} catch (Exception e) {
   System.out.println("Impossible IOException occurs: " + e.toString());
   e.printStackTrace();
}
```

Далее прочитаем данные так, как они были записаны, все значения прочитаны корректно.

Листинг 1.63: Использование Data Stream (шаг 2)

```
try {
   byte[] bytes = out.toByteArray();
   InputStream in = new ByteArrayInputStream(bytes);
   DataInputStream inData = new DataInputStream(in);

   System.out.println("Reading in the correct sequence: ");
   System.out.println("readByte: " + inData.readByte());
   System.out.println("readInt: " + inData.readInt());
   System.out.println("readLong: " + inData.readLong());
   System.out.println("readDouble: " + inData.readDouble());
   inData.close();
} catch (Exception e) {
   System.out.println("Impossible IOException occurs: " + e.toString());
   e.printStackTrace();
}
```

```
Reading in the correct sequence:
readByte: -128
readInt: 128
readLong: 128
readDouble: 128.0
```

Затем, всё ломаем и видим, что всё послушно сломалось.

Листинг 1.64: Использование Data Stream (шаг 3)

```
try {
   byte[] bytes = out.toByteArray();
   InputStream in = new ByteArrayInputStream(bytes);
   DataInputStream inData = new DataInputStream(in);

```

```
        System.out.println("Reading in a modified sequence:");
        System.out.println("readInt: " + inData.readInt());
        System.out.println("readDouble: " + inData.readDouble());
        System.out.println("readLong: " + inData.readLong());

        inData.close();
    } catch (Exception e) {
        System.out.println("Impossible IOException occurs: " + e.toString());
        e.printStackTrace();
    }
```

```
Reading in a modified sequence:
readInt: -2147483648
readDouble: -0.0
readLong: -9205252085229027328
```

Ещё сложнее `ObjectInputStream` и `ObjectOutputStream`. Для объектов процесс преобразования в последовательность байт и обратно организован несколько сложнее – объекты имеют различную структуру, хранят ссылки на другие объекты и т.д. Поэтому такая процедура получила специальное название – сериализация (serialization), обратное действие, – то есть воссоздание объекта из последовательности байт – десериализация.

Классы Reader и Writer

Рассмотренные классы – наследники `InputStream` и `OutputStream` – работают с байтовыми данными. Если с их помощью записывать или считывать текст, то сначала необходимо сопоставить каждому символу его числовой код. Такое соответствие называется кодировкой. Java предоставляет классы, практически полностью дублирующие байтовые потоки по функциональности, но называющиеся `Reader` и `Writer`, соответственно. Различия между байтовыми и символьными классами весьма незначительны. Классы-мосты `InputStreamReader` и `OutputStreamWriter` при преобразовании символов также используют некоторую кодировку.

Задания для самопроверки

1. Возможно ли чтение совершенно случайного байта данных из объекта `BufferedInputStream`?
2. Возможно ли чтение совершенно случайного байта данных из потока, к которому подключен объект `BufferedInputStream`?

1.5.4. `java.nio` и `nio2`

Основное отличие между двумя подходами к организации ввода/вывода в том, что Java IO является потокоориентированным, а Java NIO – буфер-ориентированным.

io vs nio

Потокоориентированный ввод-вывод подразумевает чтение или запись из потока и в поток одного или нескольких байт в единицу времени поочередно. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если есть необходимость произвести подобные манипуляции, придётся сначала кэшировать данные в буфере.

Подход, на котором основан Java NIO немного отличается. Данные считываются в буфер для последующей обработки. Становится возможно двигаться по буферу вперед и назад. Это дает больше гибкости при обработке данных. В то же время, появляется необходимость проверять содержит ли буфер необходимый для корректной обработки объем данных. Также необходимо следить, чтобы при чтении данных в буфер не были уничтожены ещё не обработанные данные, находящиеся в буфере.

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения вызывается `read()` или `write()` метод любого класса из пакета `java.io.*`, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения (thread) в данный момент не может делать ничего другого. Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет.

Каналы – это логические порталы, через которые осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода, данные, которые вы хотите отправить, помещаются в буфер, а он передается в канал. При вводе, данные из канала помещаются в предоставленный вами буфер.

Также, в Java NIO появилась возможность создать поток, который будет знать, какой канал готов для записи и чтения данных и может обрабатывать этот конкретный канал. Данные считываются в буфер для последующей обработки.

Листинг 1.65: Использование буфера и канала

```
try (RandomAccessFile catFile = new RandomAccessFile("cat.txt", "rw")) {
   FileChannel inChannel = catFile.getChannel();
   ByteBuffer buf = ByteBuffer.allocate(100);
   int bytesRead = inChannel.read(buf);

   while (bytesRead != -1) {
      System.out.println("Read " + bytesRead + " bytes");
      // Set read mode
      buf.flip();
      while (buf.hasRemaining()) {
         System.out.print((char) buf.get());
      }

      buf.clear();
      bytesRead = inChannel.read(buf);
   }
} catch (IOException e) { e.printStackTrace(); }
```

Подготовив всё необходимое, приложение прочитало данные в буфер, сохранив число прочитанных байт, а затем прочитанные байты посимвольно были выведены в консоль. Для чтения

данных из файла используется файловый канал. Объект файлового канала может быть создан только вызовом метода `getChannel()` для файлового объекта, поскольку нельзя напрямую создать объект файлового канала. При этом, `FileChannel` нельзя переключить в неблокирующий режим.

1.5.5. String

Класс String отвечает за создание строк, состоящих из символов. Если быть точнее, заглянув в реализацию и посмотрев способ их хранения, то строки (до Java 9) представляют собой массив символов

```
private final char value[];
```

а начиная с Java 9 строки хранятся как массив байт.

```
private final byte[] value;
```

Данный материал в первую очередь направлен на Java 1.8. В отличие от других языков программирования, где символьные строки представлены последовательностью символов, в Java они являются объектами класса `String`. В результате создания объекта типа String получается неизменяемая символьная строка, т.е. невозможно изменить символы имеющейся строки. При любом изменении строки создается новый объект типа `String`, содержащий все изменения. А значит у String есть две фундаментальные особенности: это immutable (неизменный) класс; это final класс (у класса String не может быть наследников).

```
import java.nio.charset.StandardCharsets;

String s1 = "Java";
String s2 = new String("Home");
String s3 = new String(new char[] { 'A', 'B', 'C' });
String s4 = new String(s3);
String s5 = new String(new byte[] { 65, 66, 67 });
String s6 = new String(new byte[] { 0, 65, 0, 66 }, StandardCharsets.UTF_16);
```

Экземпляр класса `String` можно создать множеством способов. Несмотря на кажущуюся простоту, класс строки – это довольно сложная структура данных и огромный набор методов.

С помощью операции конкатенации, которая записывается, как знак +, можно соединять две символьные строки, порождая новый объект типа `String`. Операции такого сцепления символьных строк можно объединять в цепочку. Строки можно сцеплять с другими типами данных, например, целыми числами. Так происходит потому, что значение типа `int` автоматически преобразуется в своё строковое представление в объекте типа `String`.

Сцепляя разные типы данных с символьными строками, следует быть внимательным, иначе можно получить неожиданные результаты.

```
String s0 = "Fifty five is " + 50 + 5; // Fifty five is 505
String s1 = 50 + 5 + " = Fifty five";  // 55 = Fifty five
```

Такой результат объясняется ассоциативностью оператора сложения. Оператор сложения «работает» слева направо, а расширение типа до максимального происходит автоматически, так если складывать целое и дробное, в результате будет дробное, если складывать любое число и строку получится строка. Однажды получив строку, дальнейшее сложение превратится в конкатенацию.

StringBuffer, StringBuilder

Поскольку строка это достаточно неповоротливо, были придуманы классы, которые позволяют ускорить работу с ними.

Если в программе планируется работа со строками в больших циклах, следует рассмотреть возможность использования `StringBuilder`.

Помните, что если что-то «тормозит», то с 90% вероятностью уже придумали способ это ускорить. Все строковые классы работают с массивами символов, но `String` делает это примитивно, выделяя каждый раз новый блок памяти под массив с длиной равной длине строки, а `StringBuilder` сразу выделяет большой блок памяти под массив, добавляет к нему элементы по индексу, а если массив заканчивается, то тогда делает массив в два раза больше, копирует в него старый, и заполняет уже его.

Разработчики Java знали, что перед программистами будут стоять задачки и посерьёзнее, чем обработка нескольких тысяч символьных строк, например, при разборе текстовых файлов, и поиске информации в электронных книгах. Поэтому придумали `StringBuilder` и `StringBuffer`. Создают они изменяемые строки и динамические ссылки на них. Их разница в том, что `StringBuilder` не потокобезопасный, и работает чуть быстрее, а `StringBuffer` – используется в многопоточных средах, но в одном потоке работает чуть медленнее.

```
String s = “”Example;
long timeStart = System.nanoTime();
for (int i = 0; i < 30000; ++i) {
   s = s + i;
}
double deltaTime = (System.nanoTime() - timeStart) * 0.000000001;
System.out.println("Delta time: " + deltaTime);

StringBuilder sb = new StringBuilder("Example");
long timeStart = System.nanoTime();
for (int i = 0; i < 100_000; ++i) {
   sb = sb.append(i);
}
double deltaTime = (System.nanoTime() - timeStart) * 0.000000001;
System.out.println("Delta time: " + deltaTime);
```

В конструктор передано начальное значение строки. То есть это всё ещё строка, но представленная другим классом

String pool

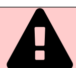
Экземпляр класса String хранится в памяти, именуемой куча (heap), но есть некоторые нюансы. Если строка, созданная при помощи конструктора хранится непосредственно в куче, то строка, созданная как строковый литерал, уже хранится в специальном месте кучи — в так называемом пуле строк (string pool). В нем сохраняются исключительно уникальные значения строковых литералов. Процесс помещения строк в пул называется интернирование (от англ. interning, внедрение, интернирование). Когда объявляется переменная типа `String` ей присваивается строковый литерал, то JVM обращается в пул строк и ищет там такое же значение. Если пул содержит необходимое значение, то компилятор просто возвращает ссылку на соответствующий адрес строки без выделения дополнительной памяти. Если значение не найдено, то новая строка будет интернирована, а ссылка на нее возвращена и присвоена переменной.

```
String cat0 = "BestCat";
String cat1 = "BestCat";
String cat2 = "Best" + "Cat";
String cat30 = "Best";
String cat3 = cat30 + "Cat";
```

В строке «Best» + «Cat» создаются два строковых объекта со значениями «Best» и «Cat», которые помещаются в пул. «Склеенные» строки образуют еще одну строку со значением «BestCat», ссылка на которую берется из пула строк (а не создается заново), т.к. она была интернирована в него ранее. Значения всех строковых литералов из данного примера известно на этапе компиляции. А если предварительно поместить один из фрагментов строки в переменную, можно «запутать» пул строк и заставить его думать, что в результате получится совсем новая строка.

```
cat0 equal to cat1? true
cat0 equal to cat2? true
cat0 equal to cat3? false
```

Когда создаётся экземпляр класса `String` с помощью оператора `new`, компилятор размещает строки в куче. При этом каждая строка, созданная таким способом, помещается в кучу (и имеет свою ссылку), даже если такое же значение уже есть в куче или в пуле строк. Это нерационально. В Java существует возможность вручную выполнить интернирование строки в пул путем вызова метода `intern()` у объекта типа `String`.



«Почему бы все строки сразу после их создания не добавлять в пул строк? Ведь это приведет к экономии памяти». Среди большого количества программистов присутствует такое заблуждение, поскольку не все учитывают дополнительные затраты виртуальной машины на процесс интернирования, а также падение производительности, связанное с аппаратными ограничениями памяти, ведь невозможно читать ячейку памяти одновременно бесконечным числом процессов.

Можно сказать, что интернирование в виде применения метода `intern()` рекомендуется не использовать. Вместо интернирования необходимо использовать дедупликацию. Если коротко, во время сборки мусора Garbage Collector проверяет живые (имеющие рабочие ссылки)

объекты в куче на возможность провести их дедупликацию. Ссылки на подходящие объекты вставляются в очередь для последующей обработки. Далее происходит попытка дедупликации каждого объекта `String` из очереди, а затем удаление из нее ссылок на объекты, на которые они ссылаются.

Задания для самопроверки

1. String – это:
   (a) объект;
   (b) примитив;
   (c) класс;
2. Строки в языке Java – это:
   (a) классы;
   (b) массивы;
   (c) объекты.

Практическое задание

1. Создать пару-тройку текстовых файлов. Для упрощения (чтобы не разбираться с кодировками) внутри файлов следует писать текст только латинскими буквами.
2. Написать метод, осуществляющий конкатенацию (объединение) переданных ей в качестве параметров файлов (не важно, в первый допишется второй или во второй первый, или файлы вовсе объединятся в какой-то третий);
3. Написать метод поиска слова внутри файла.

Термины, определения и сокращения

`finally`	часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.
`throws`	ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.
`throw`	оператор, активирующий (выбрасывающий) объект исключения.
`try...catch`	двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.
BIOS	(англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.
CI	(англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.
CLI	(англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».
cp1251	набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.
Docker	программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

2. Java Development Kit

2.1. Инструментарий: GUI – графический интерфейс пользователя

В предыдущей главе

- ООП;
- процедурное программирование;
- исключения;
- устройство языка Java;
- устройство платформы для создания и запуска приложений на JVM-языках;
- базовые средства ввода-вывода;
- базовые терминальные приложения;
- алгоритмические задачи, не требующие сложных программных решений.

В этом разделе

Интерфейс пользователя – это важно, поскольку это именно то, что видят пользователи в первую очередь.

Это самое сложное занятие начального этапа, но не потому что оно содержит очень сложную информацию, а потому что заставит под необычным углом взглянуть на те принципы ООП, которые уже известны.

Будут рассмотрены вопросы создания окон, менеджеров размещений, элементов графического интерфейса, и обработчиков событий.
- Swing;
- Асинхронность;
- Параллельность;
- Окно;
- Компонент;
- Компоновщик;
- Панель;
- События;
- Обработчик.

2.1.1. Почему именно Swing?

Вместо вступления следует кратко ответить на самый популярный вопрос.

- ✖ это популярный и современный фреймворк;
- ✖ пригодится любому программисту на Java;
- ✔ поможет лучше понять ООП;
- ✔ работа композиции из объектов;
- ✔ обмен информацией между объектами;
- ✔ явно использует ссылочную природу данных;
- ✔ улучшает запоминание базовых взаимосвязей объектов;
- ✔ без искусственных примеров (в результате будет разработана простая игра, крестики-нолики с графическим интерфейсом).

> **?** Почему не JavaFX? Фреймворк был выведен из стандартной библиотеки языка, начиная с Java 9, и достаточно сложен для базового знакомства с графическими библиотеками.
>
> **?** Почему не LibGDX? Фреймворк является надстройкой над Swing, объясняя его всё равно необходимо будет объяснять Swing/AWT.
>
> **!** Intellij IDEA — написана на Swing.

2.1.2. JFrame: Главный класс окна

Создание окна

В этом разделе происходит изучение программирования и ООП, на примере и с использованием фреймворка Swing, поэтому необходимо сосредоточиться на объектах, их свойствах и взаимосвязях.

> Изучение фреймворка Swing – не основная задача раздела, поэтому важно помнить, что не нужно зазубривать все названия всех компонентов.

Окно графического интерфейса, как и любая другая программа – это класс и объекты. В листинге 2.1 создаётся новый класс с названием `GameWindow`. Для начала, необходимо получить доступ к методам, содержащимся в библиотеке, для этого применяется наследование от класса `JFrame`, и создаётся конструктор.

Листинг 2.1: Класс окна

```
package ru.gb.jdk.one.online;
import javax.swing.*;
public class GameWindow extends JFrame {
    GameWindow() {
        //...
    }
}
```

В основном классе программы просто создаётся новый объект вновь описанного класса с окном. С точки зрения программирования и ООП не происходит ничего значительно отличающегося от котиков и пёсиков. Если запустить получившееся приложение, оно должно запуститься, и сразу завершиться, это признак верно написанного кода.

Листинг 2.2: Создание нового окна

```
package ru.gb.jdk.one.online;
public class Main {
   public static void main(String[] args) {
      new GameWindow();
   }
}
```

Закрытие окна, завершение приложения

Бо́льшая часть свойств окна не меняется за всё время существования окна и задаётся в конструкторе, то есть окно при создании будет наделено какими-то свойствами, которые в некоторых случаях можно будет поменять во время работы приложения. Самое не очевидное на первый взгляд то, что в Swing при нажатии на крестик в углу окна программа не завершается. По умолчанию, все создаваемые окна – невидимые. Это сделано потому что есть возможность создавать сколько угодно окон для приложения и такое поведение значительно снижает риск неожиданного поведения.

Все окна по умолчанию невидимые. Нажатие на крестик по умолчанию делает окно невидимым, а не завершает программу.

Для того, чтобы программа закрылась, необходимо принять решение, какое окно в ней будет главным. Чтобы программа завершалась при закрытии главного окна (обычно, это первое, что делается при создании одно оконных приложений) экземпляру JFrame устанавливается свойство `DefaultCloseOperation`. То есть устанавливается, что нужно сделать, когда это (главное) окно закроется. В это свойство записывается константа EXIT_ON_CLOSE. Если этого не сделать, то будет использовано поведение по умолчанию, окно сделается невидимым, и приложение не завершится.

Листинг 2.3: Завершение приложения по закрытию окна

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Свойства окна

Добавив констант[1] с шириной, высотой и положением окна по осям относительно рабочего стола, становится возможным настроить размеры и положение окна в конструкторе.

Листинг 2.4: Базовые свойства окна

```
private static final int WINDOW_HEIGHT = 555;
```

[1]Считается хорошим тоном не держать в коде никаких «магических цифр», чтобы не думать, что они значат и почему они именно такие, и что будет если их поменять.

```
2  private static final int WINDOW_WIDTH = 507;
3  private static final int WINDOW_POSX = 800;
4  private static final int WINDOW_POSY = 300;
5
6  GameWindow() {
7      setDefaultCloseOperation(EXIT_ON_CLOSE);
8      setLocation(WINDOW_POSX, WINDOW_POSY);
9      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
10
11     setVisible(true);
12 }
```

Установка всех начальных свойств окна осуществляется вызовом соответствующих сеттеров в конструкторе. По умолчанию окно – невидимое, поэтому для его демонстрации в конструкторе вызывается метод `setVisible` с передаваемым аргументом `true`.

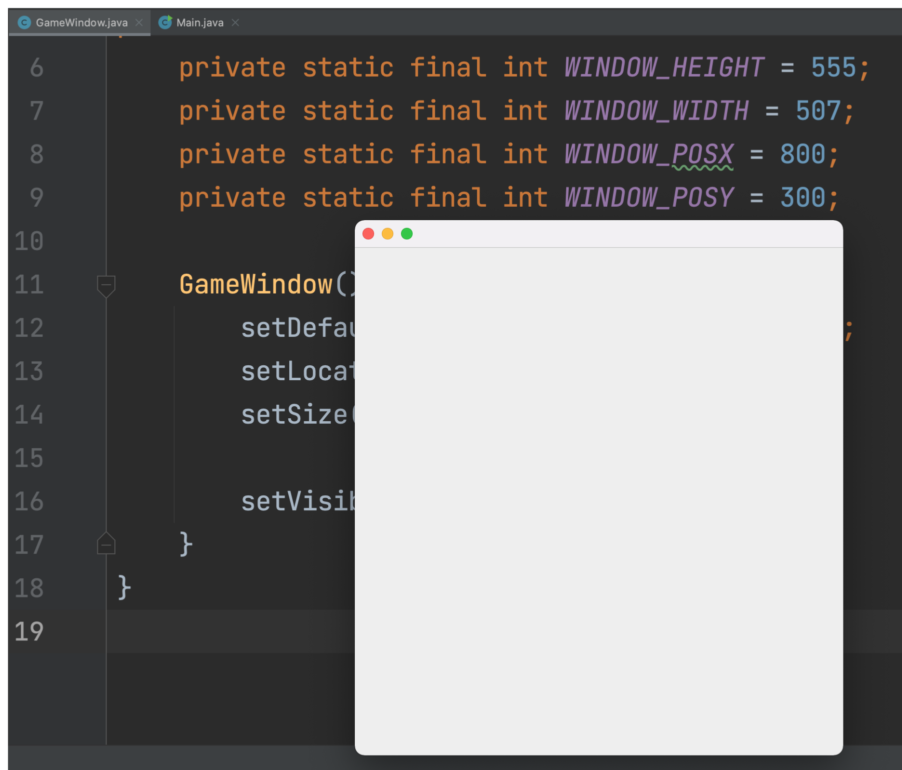


Рис. 2.1: Пустое окно указанного размера

Окно – это всегда *отдельный поток программы*, внутри которого работает бесконечный цикл. В объекте окна существует очередь сообщений, которую цикл опрашивает и выполняет.

Листинг 2.5: Пример сообщения, показывающего многопоточность

```
1  public static void main(String[] args) {
2      new GameWindow();
3      System.out.println("Method main() is over");
4  }
```

Запустив программу и внимательно изучив результат, очевидно, что окно создалось, в консоли видно, что работа метода `main` закончилась, а окно всё равно выполняется, его, при жела-

нии, можно подвигать, изменить размер и так далее. Это и есть наглядная демонстрация *многопоточности*. Таким образом, получается, что когда создаётся новое окно – нет необходимости его ни в какой контейнер помещать, ни думать, как оно будет взаимодействовать с пользователем, оно создастся и будет жить своей жизнью. **Инкапсуляция**. Если появится необходимость что-то ещё выполнить в методе `main`, запрета на написание действий не существует и действия будут выполняться *параллельно*, *асинхронно*.

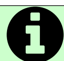
Вопросы для самопроверки

1. Почему в классе `GameWindow` доступны методы фреймворка?
   (a) Из-за устройства фреймворка Swing;
   (b) из-за наследования от JFrame;
   (c) из-за импорта классов Swing.
2. Чтобы создать пустое окно в программе нужно
   (a) импортировать библиотеку Swing;
   (b) создать класс MainWindow;
   (c) создать класс-наследник JFrame.
3. Свойства окна, такие как размер и заголовок возможно задать
   (a) написанием методов в классе-наследнике;
   (b) вызовом методов в конструкторе;
   (c) созданием констант в первых строках класса.

2.1.3. Компоненты окна

Кнопка

Для того, чтобы точно ничего не перепутать в процессе разработки, возможно придать окну больше индивидуальности, задав заголовок и запретив пользователю изменять его размеры, для игры в крестики-нолики это будет важно, чтобы красиво отображалось поле. Для этого вызовем методы `setTitle()` и `setResizeable()`, соответственно.


Элементы графического интерфейса – это хорошо знакомые кнопки, текстовые поля, надписи (лейблы), и тому подобные.

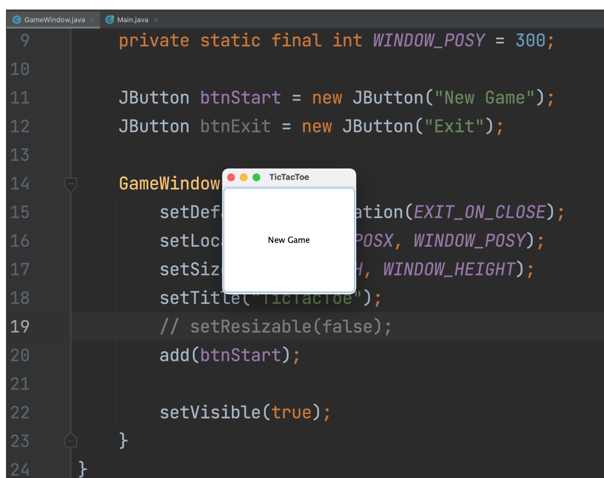
За кнопки отвечает класс `JButton`, при создании экземпляра есть возможность сразу в конструкторе задать надпись, которая будет отображаться на кнопке. Сразу создадим несколько кнопок, например, «выход». Кнопки недостаточно просто создать, поскольку неизвестно, где они должны находиться. Одну из созданных кнопок добавим на окно – для этого внутри конструктора необходимо воспользоваться методом `add()`, который требует в качестве аргумента передать ему какой-то `Component`. Все кнопки, лейблы и прочие элементы интерфейса – это наследники класса `Component`.

Листинг 2.6: Компонент «Кнопка»

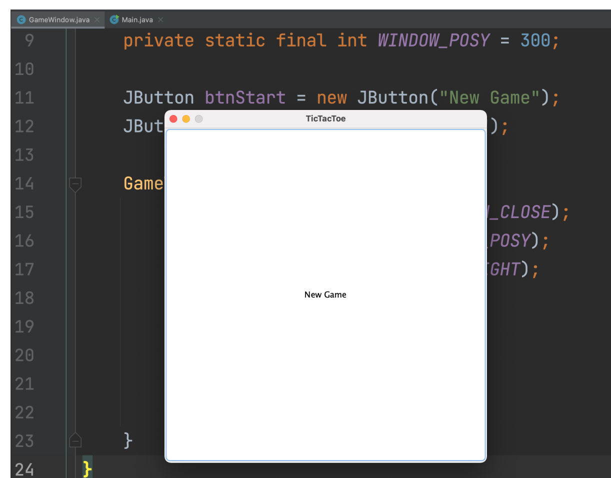
```
JButton btnStart = new JButton("New Game");
JButton btnExit = new JButton("Exit");

GameWindow() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
 6        setLocation(WINDOW_POSX, WINDOW_POSY);
 7        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
 8        setTitle("TicTacToe");
 9        setResizable(false);
10        add(btnStart);
11
12        setVisible(true);
13    }
```

После добавления в конструкторе кнопки, при запуске приложения видно, что она заняла всё окно приложения. Если убрать вызов метода `setResizeable()`, то также возможно удостовериться, что при изменении размеров окна, размер кнопки также будет меняться.



(a)



(b)

Рис. 2.2: Изменение размеров кнопки в следствие изменения размеров окна

При попытке добавить вторую кнопку на это же окно очевидно (рис. 2.3), что вторая кнопка полностью перекрыла первую.

Листинг 2.7: Вторая кнопка

```
1  add(btnExit);
```

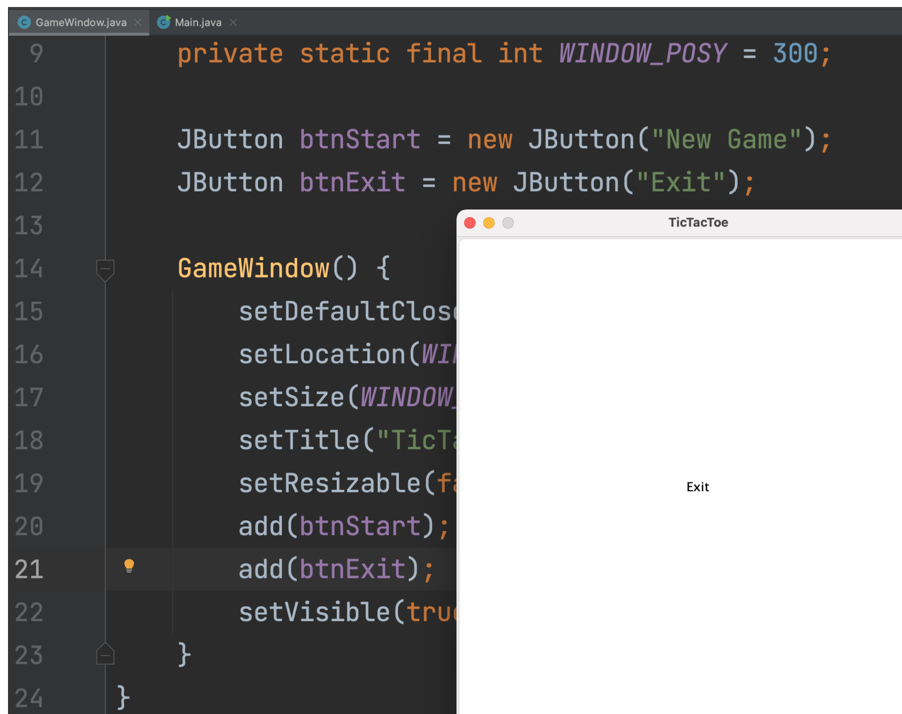


Рис. 2.3: Перекрытие компонентов

Компоновщики (менеджеры размещений)

Перекрытие происходит из-за использования компоновщика, или, как их ещё называют, менеджера размещений.

Менеджеры размещений нужны для того, чтобы не думать каждый раз о том, как изменится размер и координаты конкретного элемента, допустим, при изменении окна и не писать сложное поведение вложенных компонентов чтобы просто отобразить то, что привычно пользователю.

Компоновщики активно используются в любом программировании графических интерфейсов в любых языках программирования, от C++ до JavaScript, потому что это достаточно удобный механизм, берущий на себя значительный пласт работы. Использование компоновщиков позволяет эффективно управлять и размещать компоненты в окне или панели пользовательского интерфейса, обеспечивая гибкость и адаптивность приложения к изменениям размеров и расположения компонентов на экране.

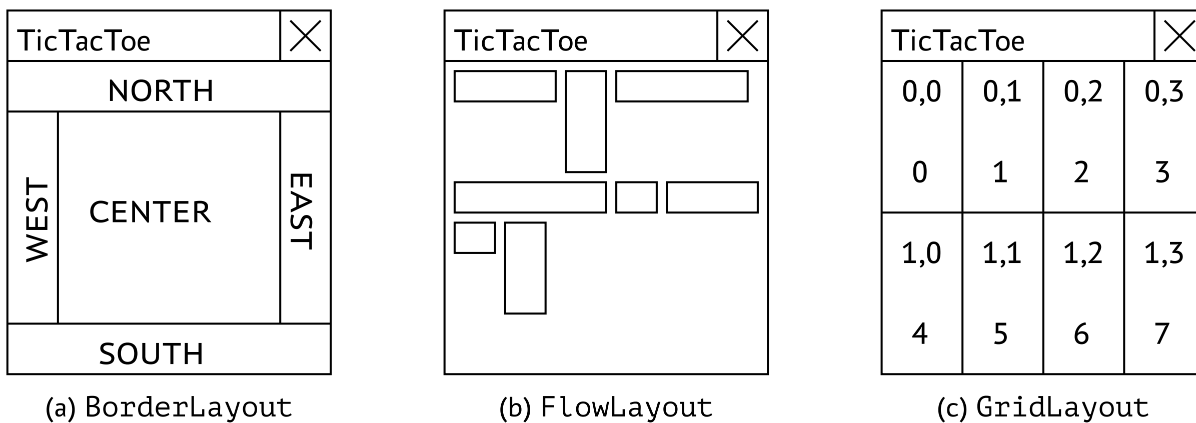
Компоновщик – это специальный объект, который помещается на некоторые (`RootPaneContainer`[2]) компоненты и осуществляет автоматическую расстановку добавляемых в него компонентов, согласно правилам. Компоновщиков существует несколько типов, каждый из которых предоставляет свои специфические возможности и алгоритмы расположения. Компоновщик выбирается в зависимости от требуемых задач и желаемого внешнего вида интерфейса.

- `BorderLayout` (по умолчанию);
- `BoxLayout`;
- `CardLayout`;
- `FlowLayout`;

[2]docs.oracle.com: Компоновщики

– GridBagLayout;
– GridLayout;
– GroupLayout;
– SpringLayout.

По умолчанию в Swing используется компоновщик BorderLayout (рис. 2.4a). Он располагает всё, что ему передаётся в центре, но также у него есть ещё четыре положения, маленькие области по краям.



(a) BorderLayout (b) FlowLayout (c) GridLayout

Рис. 2.4: Популярные менеджеры размещений

Если какая-то область не занята компонентом, она автоматически уменьшается до нулевых размеров, оставляя место другим компонентам. Поэтому, если необходимо какой-то компонент расположить не в центре, это нужно явно указать при добавлении. На первый взгляд, это немного не очевидно, поэтому лучше запомнить, что при добавлении надо указать ещё один параметр, константу, например, BorderLayout.SOUTH. FlowLayout будет располагать элементы друг за другом слева направо, сверху вниз. Компоновщик-сетка GridLayout при создании принимает на вход число строк и столбцов и располагает компоненты в получившейся сетке.

Основная идея, которую надо понять, это не названия компоновщиков, а то, что в Swing вся работа происходит через компоновщики – Layout, которые каждый по-своему располагают элементы в окне.

Панель для размещения JPanel

Разнообразие требований к разработке графических интерфейсов может привести к необходимости создания бесконечного числа компоновщиков. Поэтому разработчики библиотеки Swing придумали использовать не только компоненты сами по себе, но и группы элементов, которые располагаются на так называемых панелях (JPanel). Главная особенность панелей в том, что внутри каждой панели возможно использовать свой собственный компоновщик. JPanel – это по умолчанию невидимый прямоугольник, на котором может находиться собственный компоновщик. Например, становится доступным создание для окна панели с кнопками, а остальное пространство оставить под другие важные вещи. В листинге 2.8 описан код создания панели, добавление её в нижнюю часть основного экрана, расположение внутри панели компоновщика и двух кнопок. Важно, что на экран добавляются не кнопки по отдельности, а компонент, на который предварительно добавили кнопки.

Листинг 2.8: Создание и применение панели

```
GameWindow() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocation(WINDOW_POSX,WINDOW_POSY);
    setSize(WINDOW_WIDTH,WINDOW_HEIGHT);
    setTitle("TicTacToe");
    setResizable(false);

    JPanel panBottom = new JPanel(new GridLayout(1, 2));
    panBottom.add(btnStart);
    panBottom.add(btnExit);
    add(panBottom, BorderLayout.SOUTH);
    setVisible(true);
}
```

`JPanel` позволяет также осуществлять рисование и взаимодействие с пользователем. Основные графические интерактивности в демонстрационном приложении будут сделаны именно на панели. Такую панель с достаточно большой функциональностью логично выделить в отдельный класс. В случае игры в крестики-нолики это будет карта поля сражения (листинг 2.9). В описании конструктора для простоты панель перекрашивается в чёрный цвет (строка 8), чтобы увидеть, что панель создаётся без ошибок.

Листинг 2.9: Создание панели с полем боя

```
package ru.gb.jdk.one.online;

import javax.swing.*;
import java.awt.*;

public class Map extends JPanel {
  Map() {
    setBackground(Color.BLACK);
  }
}
```

Естественно, панель также недостаточно просто создать (листинг 2.10, строка 8), но нужно её куда-то разместить. Например, на основной экран (строка 14). Поскольку не была указана сторона экрана, панель заняла всё свободное место на окне, кроме юга, где расположилась панель с кнопками.

Листинг 2.10: Добавление панели с полем боя

```
GameWindow() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocation(WINDOW_POSX, WINDOW_POSY);
    setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    setTitle("TicTacToe");
    setResizable(false);

    Map map = new Map();

    JPanel panBottom = new JPanel(new GridLayout(1, 2));
    panBottom.add(btnStart);
```

```
12     panBottom.add(btnExit);
13     add(panBottom, BorderLayout.SOUTH);
14     add(map);
15     setVisible(true);
16 }
```

2.1.4. Многооконное приложение, взаимосвязи

Структура

Полученных знаний достаточно, чтобы начать описывать так называемую бизнес-логику. Созданная панель Map будет выполнять функции поля боя, поэтому логично расположить в ней метод startNewGame(), начинающий новую игру. В качестве параметров метод должен принимать какие-то начальные настройки самой игры. Например, будут два режима игры, компьютер против игрока и игрок против игрока, размер поля, и сразу не будем привязываться к квадратному полю 3x3, для полей больше, чем 3x3 понадобится выигрышная длина, то есть число крестиков или ноликов, расположенных подряд на одной прямой для победы той или иной стороны. В теле метода сразу будет установлена так называемая заглушка, чтобы знать, что метод вызывается и все параметры передаются верно.

Листинг 2.11: Метод начала новой игры

```
1 void startNewGame(int mode, int fSzX, int fSzY, int wLen) {
2     System.out.printf("Mode: %d;\nSize: x=%d, y=%d;\nWin Length: %d",
3             mode, fSzX, fSzY, wLen);
4 }
```

Если сразу описать архитектуру проекта, его будет проще наполнять логикой и расширять, чем если писать последовательно, удерживая общую картину в голове. Итоговое приложение приложение будет работать в двух окнах: первое – стартовое, где будут задаваться настройки поля и производиться выбор режима игры; второе – основное, где будет происходить собственно игра. Основное окно уже написано, и при его закрытии происходит выход из программы. Для создания второго окна необходимо написать ещё один класс, названный, например, `SettingsWindow`, наследник `JFrame`. Конструктор второго окна будет принимать экземпляр игрового окна. В первую очередь это сделано для передачи параметров игры, а во-вторых, чтобы красиво отцентрировать его относительно основного.

Листинг 2.12: Заготовка окна настроек новой игры игры

```
1 package ru.gb.jdk.one.online;
2 
3 import javax.swing.*;
4 
5 public class SettingsWindow extends JFrame {
6     SettingsWindow(GameWindow gameWindow) {
7 
8     }
9 }
```

В основном окне `GameWindow` понадобится два поля, одно класса `SettingsWindow` что-

бы иметь возможность экземпляр этого окна показывать когда появится необходимость и второе – это панель Map. В основном окне при создании экземпляра окна настроек в него передаётся this.

> Обратите внимание, на этот способ применять this, когда неоюходимо передать в метод ссылку на объект, который вызывает этот метод, фактически, основное окно передаёт себя.

Листинг 2.13: Создание окна настроек в основном окне

```
Map map;
SettingsWindow settings;

GameWindow() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocation(WINDOW_POSX, WINDOW_POSY);
    setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    setTitle("TicTacToe");
    setResizable(false);

    map = new Map();
    settings = new SettingsWindow(this);
    // ...
}
```

На рисунке 2.5 первый черновик диаграммы классов разрабатываемого приложения[3]. Создание идеальной диаграммы классов или модели данных не входит в цели курса, более важно понятное объяснение того, что в данный момент программируется. Буквами F обозначены экземпляры JFrame, буквой P Jpanel, а A это Application, то есть основной класс приложения. На диаграмме видно, что основное приложение создаёт основное окно, на которое добавлена панель Map и которое время от времени будет обращаться к SettingsWindow за настройками новой игры.

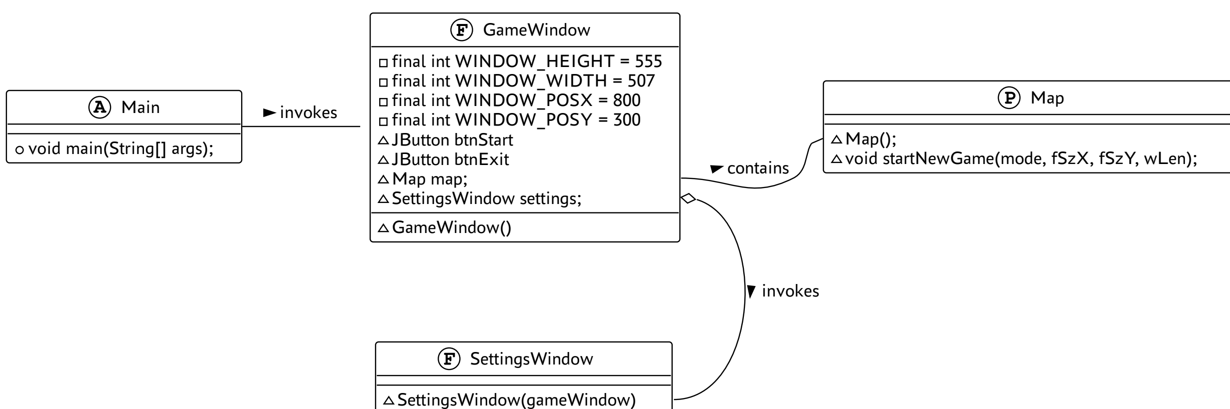


Рис. 2.5: Диаграмма классов приложения

[3]Исходный код PlantUML

Окно с настройками игры и обработчики кнопок

Окно настроек игры на данный момент будет представлено одной кнопкой старта игры, вызывающей метод старта игры с одним зафиксированным набором настроек – игра против компьютера, поле 3х3, чтобы выиграть необходимо собрать 3 крестика (или нолика) подряд.

В данный момент окно создаётся в координатах (0,0) и имеет размер (0,0), то есть в левом верхнем углу экрана видно только кнопки свернуть, развернуть, закрыть. В конструкторе окна задаются его размеры и то, что его местоположение должно быть относительным главному окну. Аналогично основному окну добавлена кнопка подтверждения правильности настроек и старта игры.

Листинг 2.14: Окно настроек новой игры

```
public class SettingsWindow extends JFrame {
    private static final int WINDOW_HEIGHT = 230;
    private static final int WINDOW_WIDTH = 350;

    JButton btnStart = new JButton("Start new game");
    SettingsWindow(GameWindow gameWindow) {
        setLocationRelativeTo(gameWindow);
        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

        add(btnStart);
    }
}
```

Далее необходимо «оживить» кнопки на окнах, это делается специальной конструкцией, синтаксис которой пока что придётся запомнить. Синтаксически всё написанное уже понятно и оговорено – у объекта кнопки `btnExit` вызывается метод добавления к этому объекту некоторого слушателя действия. Какое может у кнопки быть самое очевидное действие? Нажатие. В аргумент метода добавления передаётся некий новый объект класса «слушатель действия», у которого переопределяется метод «действие произошло». Кнопка старта игры будет делать видимым окно с будущими настройками. В листинге 2.15 показаны обработчики кнопок старта новой игры и завершения приложения, находящихся на основном окне программы. Эти обработчики необходимо поместить внутрь конструктора основного окна.

Листинг 2.15: Обработчики нажатий на кнопки основного окна

```
btnExit.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});

btnStart.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        settings.setVisible(true);
    }
});
```

Последовательность выполнения программы

В основном окне понадобится метод, инициализирующий новое игровое поле, поскольку прямой вызов по кнопке старта новой игры на окне настроек метода в панели основного окна противоречит инкапсуляции.

Зачем мы так делаем, казалось бы усложняем? Но нет: панель находится на основном окне, а кнопка начала игры будет находиться на окне настроек, которое не может «знать», какие на основном окне есть панели. Или может оказаться, что дальше нет никакого интерфейса, а игра происходит по сети.

В этом и есть суть ООП, когда один объект максимально отделён от другого, и каждому из них вообще не важно, как реализован другой.

Соответственно, когда в окне настроек нажата кнопка «начать игру», обработчик вызывает метод главного окна, а главное окно в свою очередь уже знает, что оно разделено на панели, и вынуждает панель Map начинать (рисунок 2.6)[4]. Для чего нужен промежуточный метод? Чтобы не делать лишних связей между классами. Это логично с точки зрения инкапсуляции. Одно окно не должно никак управлять панелью на другом окне.

Листинг 2.16: Обработчик кнопки старта новой игры

```
SettingsWindow(GameWindow gameWindow) {
   setLocationRelativeTo(gameWindow);
   setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
   btnStart.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
         gameWindow.startNewGame(0, 3, 3, 3);
         setVisible(false);
      }
   });
   add(btnStart);
}
```

В окне настроек описан обработчик нажатия на единственную кнопку, из этого обработчика вызывается единственный доступный метод – «старт новой игры» на основном окне. По факту нажатия, также, целесообразно спрятать окно настроек. Из метода основного класса `startNewGame()` вызывается `map.startNewGame()` класса мэп.

Листинг 2.17: Цепочка вызовов методов старта новой игры

```
void startNewGame(int mode, int fSzX, int fSzY, int wLen) {
   map.startNewGame(mode, fSzX, fSzY, wLen);
}
```

Ещё раз цепочка вызовов (рис. 2.6):

1. основное окно делает окно настроек видимым;
2. окно настроек говорит основному, что пора начинать игру;
3. основное окно в свою очередь знает, как именно надо игру начинать и просит панель стартовать.

[4]Исходный код диаграммы в PlantUML

Если всё сделано верно, в терминале появится вывод из заглушки на панели `Map`.

```
Mode: 0;
Size: x=3, y=3;
Win Length: 3
```

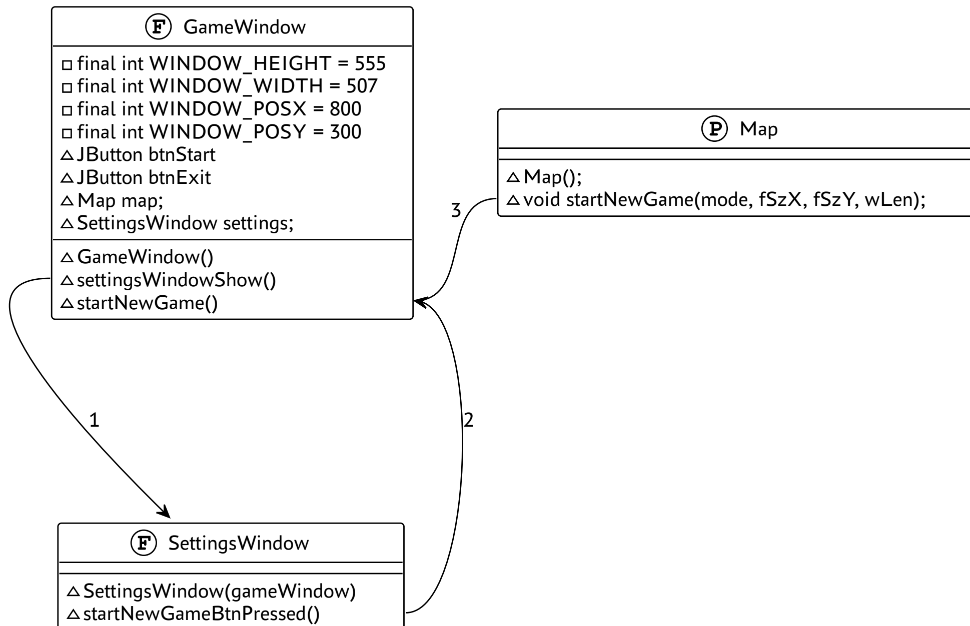


Рис. 2.6: Цепочка вызовов при старте новой игры

Вопросы для самопроверки

1. Менеджер размещений - это
   (a) сотрудник, занимающийся разработкой интерфейса;
   (b) объект, выполняющий расстановку компонентов на окне приложения;
   (c) механизм, проверяющий возможность отображения окна в ОС.
2. Экземпляр JPanel позволяет
   (a) применять комбинации из компоновщиков
   (b) добавить к интерфейсу больше компонентов
   (c) создавать группы компонентов
   (d) всё вышеперечисленное
3. Для выполнения кода по нажатию кнопки на интерфейсе нужно
   (a) создать обработчик кнопки и вписать код в него
   (b) переопределить метод нажатия у компонента кнопки
   (c) использовать специальный класс “слушателя” кнопок

2.1.5. Основная панель с игрой

Рисование

Далее всё будет происходить на панели с полем для игры. Для рисования самой панели фреймворком определён метод `paintComponent()`. Этот метод вызывается фреймворком когда что-то происходит, например, когда основное окно перекрывается другим, перемещается на

другой экран, или если его развернуть из свёрнутого состояния, вызывается он гораздо реже, чем это необходимо для логики игры. Для описания игрового процесса необходимо перерисовывать компонент по каждому клику мышкой и по каждому действию оппонента.

Важно помнить, что метод `paintComponent()` не следует напрямую вызывать из кода. Этот метод должен вызываться только фреймворком. Для того чтобы запросить у фреймворка вызов этого метода тоже есть специальный метод.

Для дальнейшей разработки важно отделить стандартный метод рисования компонента от пользовательского рисования на этом компоненте, так называемую бизнес-логику. Для этого будет создан ещё один метод `void render(Graphics g)`, который будет вызываться из переопределённого `paintComponent()`. из самого `paintComponent()` вызов метода родительского класса удалять не следует, поскольку там, скорее всего, происходит что-то важное. Для вызова же метода фреймворка, необходимо в нужный момент сказать фреймворку что требуется перерисовать панель, фреймворк поставит метод `paintComponent()` в очередь сообщений окна, и когда очередь дойдёт до выполнения этого метода – окно выполнит перерисовку.

Листинг 2.18: Отделение бизнес-логики рисования

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    render(g);
}

private void render(Graphics g) { }
```

Это действие полностью асинхронно и *косвенно* зависит от наших вызовов

Чтобы рисовать нужен объект класса `Graphics`, который умеет рисовать геометрические фигуры, линии, текст и тому подобное. Чтобы нарисовать поле для игры понадобится ширина и высота поля в пикселях. Их возможно узнать из свойств панели – ширины и высоты. Всё, что связано с размерами лучше вынести в переменные объекта, поскольку они понадобятся в других методах. Помимо ширины и высоты понадобятся переменные, в которых будет храниться высота и ширина каждой ячейки. Размеры каждой ячейки пригодятся для создания отступа одной линии от другой. Далее циклически делаются отступы и рисуются горизонтальные и вертикальные линии.

Листинг 2.19: Разлиновка поля для игры

```
private int panelWidth;
private int panelHeight;
private int cellHeight;
private int cellWidth;

private void render(Graphics g) {
    panelWidth = getWidth();
    panelHeight = getHeight();
    cellHeight = panelHeight / 3;
    cellWidth = panelWidth / 3;
```

```
    g.setColor(Color.BLACK);
    for (int h = 0; h < 3; h++) {
        int y = h * cellHeight;
        g.drawLine(0, y, panelWidth, y);
    }
    for (int w = 0; w < 3; w++) {
        int x = w * cellHeight;
        g.drawLine(x, 0, x, panelHeight);
    }
    repaint();
}
```

У многих разработчиков, в зависимости от используемой операционной системы после запуска этого кода не полностью или вовсе не рисуется разлиновка, это происходит из-за асинхронности рисования, скорее всего метод с линиями отработал позже того, как Swing нарисовал панель.

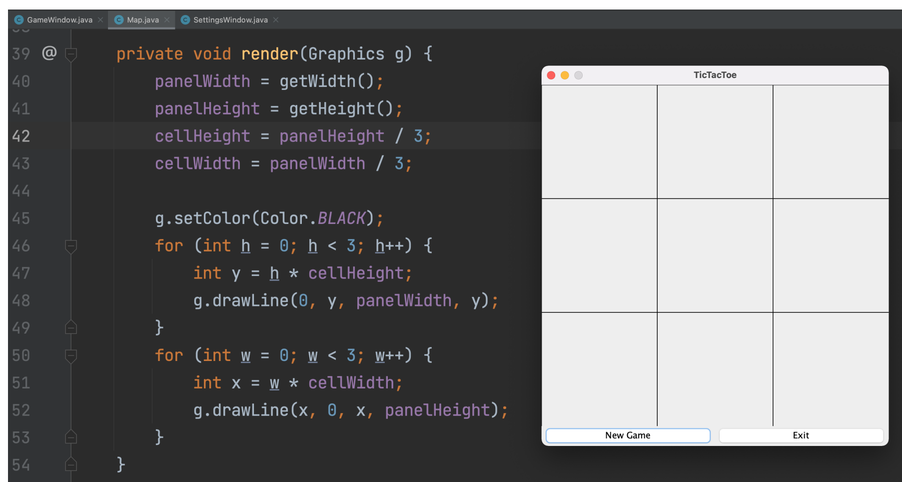


Рис. 2.7: Результат разлиновки поля для игры

Чтобы всё увидеть, необходимо заставить компонент панели полностью перерисоваться. Это делается вызовом метода `repaint()` из метода старта новой игры.

Обработчик мышки

Обработчик действий мышки очень похож на те обработчики, которые уже написаны. В конструкторе панели описывается метод добавления слушателя, в котором переопределяется метод `mouseReleased()`, то есть для приложения важно когда пользователь отпустит кнопку и аналогично методу отрисовки следует сразу отделить обработчик от основной исполняемой логики.

Листинг 2.20: Обработчик отпускания кнопки мышки

```
Map() {
    addMouseListener(new MouseAdapter() {
        @Override
        public void mouseReleased(MouseEvent e) {
```

```
        update(e);
      }
   });
}
```

Внутри метода обновления также принудительно вызывается метод перерисовки компонента, чтобы получился игровой цикл: старт – отрисовка – клик мыши – отрисовка – клик – отрисовка...

Листинг 2.21: Обработчик отпускания кнопки мышки

```
private void update(MouseEvent e) {
   int cellX = e.getX() / cellWidth;
   int cellY = e.getY() / cellHeight;
   System.out.printf("x=%d, y=%d\n", cellX, cellY);
   repaint();
}
```

В методе обновления из объекта `MouseEvent` получаются координаты клика, делятся на размер ячейки и тем самым получается номер ячейки, в которую произошёл клик.

Логика игры

Поскольку лекция про графические интерфейсы и ООП, а все используемые конструкции примитивны, код логики игры будет приведён без подробных пояснений.

Для работы понадобится генератор псевдослучайных чисел, символы, которыми будет обозначаться на поле игрок, компьютер и пустая ячейка, собственно поле и его размеры. Размеры – на будущее.

Листинг 2.22: Субъекты игры

```
private static final Random RANDOM = new Random();
private final int HUMAN_DOT = 1;
private final int AI_DOT = 2;
private final int EMPTY_DOT = 0;
private int fieldSizeY = 3;
private int fieldSizeX = 3;
private char[][] field;
```

Метод инициализации поля – создаётся новый массив и заполняется пустыми символами. Его вызов логично сразу разместить в метод старта новой игры.

Листинг 2.23: Инициализация карты

```
private void initMap() {
   fieldSizeY = 3;
   fieldSizeX = 3;
   field = new char[fieldSizeY][fieldSizeX];
   for (int i = 0; i < fieldSizeY; i++) {
      for (int j = 0; j < fieldSizeX; j++) {
         field[i][j] = EMPTY_DOT;
      }
   }
```

```
}
```

Когда кто-то (игрок или компьютер) будет совершать ход, будет важно, попал ли игрок в какую-то ячейку поля и пустота ли она, потому что нельзя ставить крестик поверх нолика и наоборот.

Листинг 2.24: Попал ли игрок в пустую ячейку и в поле

```
private boolean isValidCell(int x, int y) {
   return x >= 0 && x < fieldSizeX && y >= 0 && y < fieldSizeY;
}

private boolean isEmptyCell(int x, int y) {
   return field[y][x] == EMPTY_DOT;
}
```

Компьютер будет очень примитивный – он будет делать ход в случайные места на карте.

Листинг 2.25: Ход компьютера

```
private void aiTurn() {
   int x, y;
   do {
      x = RANDOM.nextInt(fieldSizeX);
      y = RANDOM.nextInt(fieldSizeY);
   } while (!isEmptyCell(x, y));
   field[y][x] = AI_DOT;
}
```

Очевидно, что учебные цели предполагают не только демонстрацию того, как надо делать, но также и демонстрацию того как делать не надо. Далее приведён код, который не следует допускать при работе над приложениями. Метод принимает на вход символ, который нужно проверить и проверяет - не победил ли он.

Листинг 2.26: Проверка победы одного из игроков

```
private boolean checkWin(char c) {
   if (field[0][0]==c && field[0][1]==c && field[0][2]==c) return true;
   if (field[1][0]==c && field[1][1]==c && field[1][2]==c) return true;
   if (field[2][0]==c && field[2][1]==c && field[2][2]==c) return true;

   if (field[0][0]==c && field[1][0]==c && field[2][0]==c) return true;
   if (field[0][1]==c && field[1][1]==c && field[2][1]==c) return true;
   if (field[0][2]==c && field[1][2]==c && field[2][2]==c) return true;

   if (field[0][0]==c && field[1][1]==c && field[2][2]==c) return true;
   if (field[0][2]==c && field[1][1]==c && field[2][0]==c) return true;
   return false;
}
```

> Всегда пишите с помощью циклов, потому что стоит захотеть изменить размер поля на 4x4 или 5x5 – размер и сложность этого метода будет расти в геометрической прогрессии.

И метод проверки поля на состояние ничьей. Ничья в крестиках-ноликах наступает, когда не победил ни игрок ни оппонент, и не осталось пустых клеток.

Листинг 2.27: Проверка на ничью

```
private boolean isMapFull() {
   for (int i = 0; i < fieldSizeY; i++) {
      for (int j = 0; j < fieldSizeX; j++) {
         if (field[i][j] == EMPTY_DOT) return false;
      }
   }
   return true;
}
```

Вся дальнейшая работа будет сконцентрирована на методах обновления игрового состояния и отрисовки игрового поля. В результате клика в ячейку необходимо проверить, валидная-ли ячейка, и можно-ли туда ходить. Если какое-то из условий не прошло, клик игнорируется, а если всё хорошо – делается ход.

Листинг 2.28: Ход игрока

```
int cellX = e.getX()/cellWidth;
int cellY = e.getY()/cellHeight;
if (!isValidCell(cellX, cellY) || !isEmptyCell(cellX, cellY)) return;
field[cellY][cellX] = HUMAN_DOT;

repaint();
```

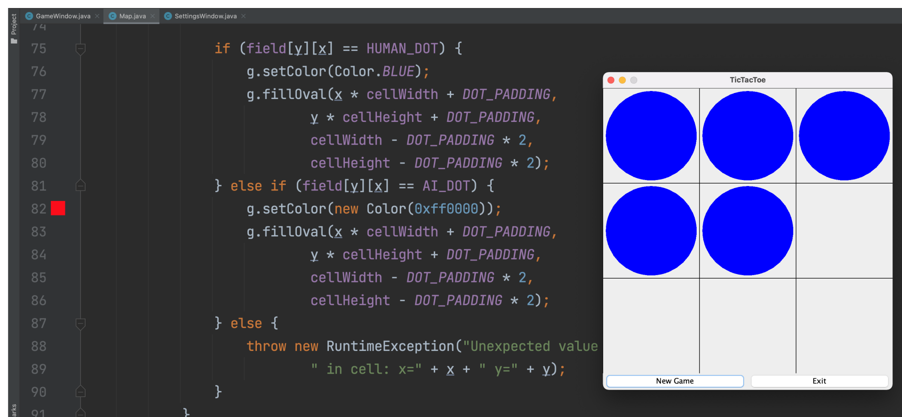
В метод отрисовки также необходимо добавить логику. Если в ячейке поля ничего нет – ничего делать не нужно, далее условие, если в ячейке крестик – что-то сделаем, если нолик – сделаем что-то другое и в противном случае выбросим исключение.

Листинг 2.29: Отрисовка поля

```
for (int y = 0; y < fieldSizeY; y++) {
   for (int x = 0; x < fieldSizeX; x++) {
      if (field[y][x] == EMPTY_DOT) continue;

      if (field[y][x] == HUMAN_DOT) {
         g.setColor(Color.BLUE);
         g.fillOval(x * cellWidth + DOT_PADDING,
               y * cellHeight + DOT_PADDING,
               cellWidth - DOT_PADDING * 2,
               cellHeight - DOT_PADDING * 2);
      } else if (field[y][x] == AI_DOT) {
         g.setColor(new Color(0xff0000));
         g.fillOval(x * cellWidth + DOT_PADDING,
               y * cellHeight + DOT_PADDING,
```

```
15                      cellWidth - DOT_PADDING * 2,
16                      cellHeight - DOT_PADDING * 2);
17             } else {
18                 throw new RuntimeException("Unexpected value " + field[y][x] +
19                         " in cell: x=" + x + " y=" + y);
20             }
21         }
22 }
```

Далее – непосредственно отрисовка. Сюда можно картинку вставлять, закрашивать квадраты, рисовать крестики и нолики. Для простоты будут рисоваться кружки. Методу объекта графики `g.fillOval()` в сигнатуре передаётся левая верхняя координата прямоугольника, в который затем будет вписан овал, его ширина и высота соответственно. Чтобы задать цвет – перед тем как рисовать необходимо изменить цвет объекта графики `g.setColor(Color.BLUE)`. Для человека далее будут рисоваться синие кружки, а для компьютера красные.



Последние приготовления

По сути, осталось сделать две вещи – описать так называемую бизнес-логику, то есть в правильном порядке вызвать методы с логикой игры, избавиться от исключений и вывести сообщение об окончании игры. Для того, чтобы вывести результат, поверх игрового поля будет выводиться сообщение.

Листинг 2.30: Состояния игрового поля

```
1 private int gameOverType;
2 private static final int STATE_DRAW = 0;
3 private static final int STATE_WIN_HUMAN = 1;
4 private static final int STATE_WIN_AI = 2;
5 
6 private static final String MSG_WIN_HUMAN = "Победил игрок!";
7 private static final String MSG_WIN_AI = "Победил компьютер!";
8 private static final String MSG_DRAW = "Ничья!";
```

В методе обновления уточняется, что когда пользователь поставил точку, необходимо проверить состояние поля на наличие победы или ничьей, дать возможность компьютеру поставить точку и сделать тоже самое.

Листинг 2.31: Логика обновления

```
// update
if (checkEndGame(HUMAN_DOT, STATE_WIN_HUMAN)) return;
aiTurn();
repaint();
if (checkEndGame(AI_DOT, STATE_WIN_AI)) return;
// end update

private boolean checkEndGame(int dot, int gameOverType) {
   if (checkWin(dot)) {
      this.gameOverType = gameOverType;
      repaint();
      return true;
   }
   if (isMapFull()) {
      this.gameOverType = STATE_DRAW;
      repaint();
      return true;
   }
   return false;
}
```

В методе рендеринга, как только поле выведено и если игра закончилась необходимо вывести сообщение с одним из вариантов исхода игры. Для упрощения также следует завести классовую переменную с признаком окончания игры. Метод окончания игры рисует тёмно серый прямоугольник с жёлтой надписью о победе одного из игроков или ничьей в зависимости от состояния.

Листинг 2.32: Отрисовка сообщения об окончании игры

```
//render
if (isGameOver) showMessageGameOver(g);
// end render

private void showMessageGameOver(Graphics g) {
   g.setColor(Color.DARK_GRAY);
   g.fillRect(0, 200, getWidth(), 70);
   g.setColor(Color.YELLOW);
   g.setFont(new Font("Times new roman", Font.BOLD, 48));
   switch (gameOverType) {
      case STATE_DRAW:
         g.drawString(MSG_DRAW, 180, getHeight() / 2); break;
      case STATE_WIN_AI:
         g.drawString(MSG_WIN_AI, 20, getHeight() / 2); break;
      case STATE_WIN_HUMAN:
         g.drawString(MSG_WIN_HUMAN, 70, getHeight() / 2); break;
      default:
         throw new RuntimeException("Unexpected gameOver state: " +
            gameOverType);
   }
}
```

Далее в листинге 2.33 приведены несколько мелких правок в методах. Прочитав текст исключения при старте приложения становится ясно, что оно возникает, когда программа не может что-то поделить на ноль. Размеры поля до их инициализации равны нулю, поэтому понадобится ещё одна булева переменная – инициализирована ли игра.

- В конструкторе панели поле не инициализировано;
- в методе обновления нет смысла обрабатывать клики по неинициализированному полю или полю на котором закончилась игра;
- при старте новой игры игра перестаёт быть законченой, а поле становится инициализированным;
- рендеринг на неинициализированном поле не имеет смысла;
- в методе проверки на победу нужно добавить присвоение истинности булевой переменной с фактом окончания игры.

Листинг 2.33: Незначительные добавления в методах панели

```
private boolean isGameOver;
private boolean isInitialized;

Map() {
    isInitialized = false;
}

private void update(MouseEvent e) {
    if (isGameOver || !isInitialized) return;
}

void startNewGame(int mode, int fSzX, int fSzY, int wLen) {
    isGameOver = false;
    isInitialized = true;
}

private void render(Graphics g) {
    if (!isInitialized) return;
}

private boolean checkEndGame(int dot, int gameOverType) {
    if (checkWin(dot)) {
        isGameOver = true;
    }
    if (isMapFull()) {
        isGameOver = true;
    }
    return false;
}
```

Результат запуска получившегося приложения представлен на рисунке 2.8.

(a) Победа компьютера

(b) Победа игрока

(c) Ничья

Рис. 2.8: Результаты игры

Практическое задание

1. Полностью разобраться с кодом.
2. Переделать проверку победы, чтобы она не была реализована просто набором условий.
3. Попробовать переписать логику проверки победы, чтобы она работала для поля 5х5 и количества фигур 4.
4. ** Доработать искусственный интеллект, чтобы он мог примитивно блокировать ходы игрока, и примитивно пытаться выиграть сам.

Термины, определения и сокращения

`finally` — часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.

`throws` — ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.

`throw` — оператор, активирующий (выбрасывающий) объект исключения.

`try...catch` — двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.

BIOS — (англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.

CI — (англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CLI — (англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».

cp1251 — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

2.2. Инструментарий: программные интерфейсы

В предыдущем разделе

В предыдущем разделе были рассмотрены графические интерфейсы пользователя.
- Создание окон,
- размещение компонентов,
- рисование элементов,
- обработка событий.

В этом разделе

Будет рассмотрено понятие и принцип работы программных интерфейсов, ключевое слово `implements`. Реализация интерфейса, реализация по умолчанию, частичная реализация интерфейса, наследование и множественное наследование интерфейсов. Отдельно будет рассмотрен принцип создания так называемых адаптеров и анонимных классов. Знания в области применения графических фреймворков будут дополнены информацией о поведении исключений на интерфейсе пользователя.
- Интерфейс;
- implements;
- Анон. класс;
- Реализация;
- Адаптер;
- Ре́ндеринг;
- SOLID;

Обычно – теория и примеры. В этом разделе повествование будет построено от практики и плохого кода к хорошему коду и теоретическому обоснованию сделанного.

2.2.1. Введение и результат

В этом разделе важно не особенно обращать внимание на то, какие именно классы и методы используются, а внимательно следить за взаимодействием и отношениями объектов, потому что интерфейсы, о которых далее планируется говорить – это механизм упрощающий и универсализирующий взаимодействия объектов.

Поначалу, код может показаться непростым, но задача будет поставлена таким образом, что без программных интерфейсов не обойтись. Почему будет сложно? Несмотря на то что принципы ООП уже известны, нужно уметь их применять.

Рис. 2.9: Результат выполнения написанного кода

В результате работы с этим разделом будет создан некий небольшой демонстрационный набросок игрового двухмерного движка, без физики, с объектами и анимацией. На рисунке 2.9 изображено окно с кружками[5]. На окне ничего не обрабатывается и практически ничего не происходит. Для реализации задуманного понадобится: окно, которое будет взаимодействовать с операционной системой, канва, на которой будет происходить рисование, и объекты, которые будут нарисованы.

2.2.2. Подготовка проекта

Основное окно

Сложное окно не нужно: константы с размерами, координатами, и конструктор здесь же в основном методе. Самое важное сейчас это то, что окно – это объект с какими-то свойствами и каким-то поведением.

Листинг 2.34: Основное окно

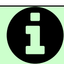
```
1  package ru.gb.jdk.two.online;
2
3  import javax.swing.*;
4
5  public class MainWindow extends JFrame {
6      private static final int POS_X = 400;
7      private static final int POS_Y = 200;
8      private static final int WINDOW_WIDTH = 800;
9      private static final int WINDOW_HEIGHT = 600;
10
11     private MainWindow() {
```

[5]Видео с демонстрацией движения

```
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setBounds(POS_X, POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT);
        setTitle("Circles");

        setVisible(true);
    }

    public static void main(String[] args) {
        new MainWindow();
    }
}
```

Канва для рисования

Для рисования будет использоваться компонент `JPanel`, наследнику панели будет дано название `MainCanvas`. Любой компонент фреймворка Swing может перерисовываться, вызывая метод `paintComponent()`. Для начала, в конструкторе панели для отработки связи компонентов следует делать что-то незначительное, например, менять цвет фона на синий.


Переопределив метод перерисовки панели не следует удалять вызов родительского метода, поскольку предполагается, что перерисовка панели происходит хорошо, но туда будет добавлена логика.

Также для универсализации дальнейших вызовов (удобства взаимодействия с канвой) следует добавить методы, возвращающие границы панели.

Листинг 2.35: Панель для канвы

```
package ru.gb.jdk.two.online;

import javax.swing.*;
import java.awt.*;

public class MainCanvas extends JPanel {
    MainCanvas() {
        setBackground(Color.BLUE);
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
    }

    public int getLeft() { return 0; }
    public int getRight() { return getWidth() - 1; }
    public int getTop() { return 0; }
    public int getBottom() { return getHeight() - 1; }
}
```

Далее необходимо расположить канву на основном окне и осуществить привязку всех действий канвы ко времени физического мира. В конструкторе основного окна создаётся переменная класса `MainCanvas` и располагается в центре.

Нет прямого запрета на написание логики будущего движка или игры в классе канвы, но это архитектурно неверное решение, ведь на канве, на которой происходит рисование, должно происходить только рисование. Подробнее в разделе 2.2.5

Исходя из принципа единой ответственности, принимается решение о том, что логика взаимодействия объектов будет описана в основном классе, а `MainCanvas` останется универсальным, чтобы иметь возможность в дальнейшем рисовать что угодно. Для этого следует описать в основном окне метод, который будет периодически вызываться канвой, например, `onDrawFrame()`. В нём будет описываться бизнес-логика. На начальном этапе, это два метода – `update()` который будет изменять состояние приложения, и `render()`, который будет отдавать команды рисующимся компонентам.

Листинг 2.36: Отделение логики

```
private MainWindow() {
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setBounds(POS_X, POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT);
    setTitle("Circles");

    MainCanvas canvas = new MainCanvas();
    add(canvas);
    setVisible(true);
}

public void onDrawFrame() {
    update();
    render();
}

private void update() { }
private void render() { }
```

Цикл отрисовки

Перерисовка канвы – это циклический процесс, и на каждой итерации `MainCanvas` должен вызывать метод `onDrawFrame()` основного класса. Для этого канве необходимо иметь ссылку на основное окно и внутри метода `paintComponent()` вызывается метод `controller.onDrawFrame()`.

Листинг 2.37: Событие отрисовки

```
public class MainCanvas extends JPanel {
    private final MainWindow controller;

    MainCanvas(MainWindow controller) {
        setBackground(Color.BLUE);
```

```
        this.controller = controller;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        controller.onDrawFrame();
    }
```

Далее, чтобы зациклить это действие, возможно два пути: самый простой – создать постоянно обновляющуюся канву, то есть в методе `paintComponent()` вызывать `repaint()` но это полностью нагрузит одно из ядер процессора только отрисовкой окна.

Листинг 2.38: Цикл отрисовки

```
public class MainCanvas extends JPanel {
    private final MainWindow controller;

    MainCanvas(MainWindow controller) {
        setBackground(Color.BLUE);
        this.controller = controller;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        controller.onDrawFrame();
        repaint();
    }
```

Второй путь – любой поток возможно заставить какое-то время поспать, для этого вызывается статический метод класса `Thread`, принимающий в качестве аргумента количество миллисекунд, которое поток должен обязательно поспать. Это даст FPS[6] близкий к 60, приемлемый для применения в цифровой технике.

Листинг 2.39: Снятие нагрузки с процессора

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    controller.onDrawFrame();
    try {
        Thread.sleep(16);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    repaint();
}
```

В результате создан бесконечный цикл отрисовки, аналогичный циклу `do-while`, который

[6]FPS, Frames per second – (англ. кадров в секунду) количество сменяемых кадров за единицу времени в кинематографе, телевидении, компьютерной графике и т. д.

сам себя заставляет крутиться с некоторой периодичностью и на каждой итерации сообщает контроллеру, что прошло около одной шестидесятой секунды.

Листинг 2.40: Изменения в основном окне

```java
private MainWindow() {
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setBounds(POS_X, POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT);
    setTitle("Circles");

    MainCanvas canvas = new MainCanvas(this);
    add(canvas);
    setVisible(true);
}
```

В код, вызывающий конструктор канвы, также требуется внести незначительные изменения. В канву необходимо передать ссылку на основное окно, на котором она находится, для этого используется ключевое слово `this`.

Такой способ применения ключевого слова `this` ещё пригодится в этом разделе.

Параметры отрисовки

Метод `onDrawFrame()` будет обновлять сцену и заставлять объекты на ней рисовать самих себя (ре́ндерить сцену). Для обновления сцены, привязанного ко времени физического мира необходимо знать дельту времени, то есть период времени, прошедший с появления предыдущего кадра.

Писать логику обновления, исходя из частоты кадра, или из того, что канва «спит» 16 миллисекунд – очень сомнительная опора, потому что поток **гарантированно** ждёт 16 миллисекунд. При этом, сколько будут выполняться остальные действия – неизвестно, так как отрисовка происходит не через фиксированные промежутки времени, а по очереди сообщений окна и под влиянием множества других факторов.

Метод `onDrawFrame()` должен принимать от канвы ряд параметров и распределять их по методам обновления и рендеринга.

Листинг 2.41: Параметры метода обновления

```java
public void onDrawFrame(MainCanvas canvas, Graphics g, float deltaTime) {
    update(canvas, deltaTime);
    render(canvas, g);
}

private void update(MainCanvas canvas, float deltaTime) {

}
private void render(MainCanvas canvas, Graphics g) {
```

```
10
11 }
```

При вычислении дельты времени важно привести все единицы измерения к единому и привычному времени, например, к секундам. Скоростью в этом случае будет «пиксель в секунду» и из метода будет отдаваться время в секундах. При обращении к контроллеру передаётся также ссылка на текущий объект канвы и объект графики.

Пока самое главное, что нужно понять об этих двух объектах – канва считает время в физическом мире и постоянно перерисовывает себя, сообщая об этом факте основному окну, а основное окно на этот факт как-то реагирует[7].

Листинг 2.42: Вычисление дельты времени между кадрами канвы

```
private long lastFrameTime;

MainCanvas(MainWindow controller) {
    this.controller = controller;
    lastFrameTime = System.nanoTime();
}

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    try {
        Thread.sleep(16);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    float deltaTime = (System.nanoTime() - lastFrameTime) * 0.000000001f;
    controller.onDrawFrame(this, g, deltaTime);
    lastFrameTime = System.nanoTime();
    repaint();
}
```

Приложение будет рисовать какие-то объекты, и не важно, будут ли это кружки, квадратики, картинки, человечки или какие-то другие объекты. Важно, чтобы у программы было описано поведение этих объектов [8].

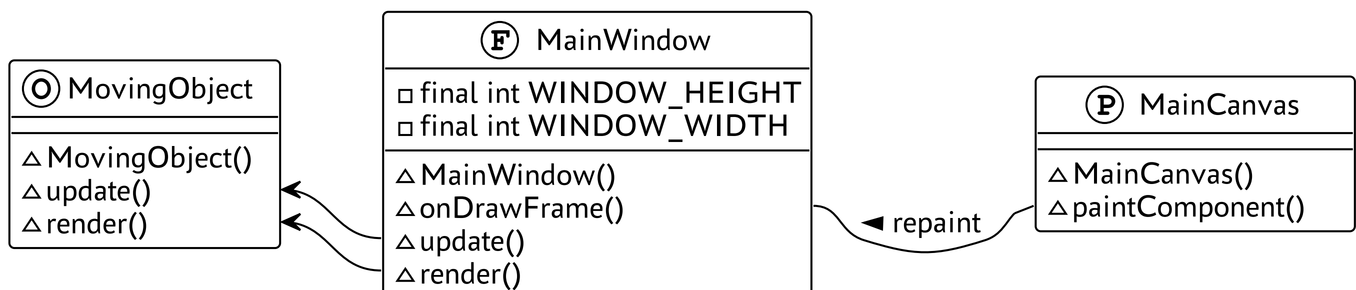


Рис. 2.10

[7]ООП вокруг этих событий тоже важно хорошо понимать – объекты передают ссылки друг на друга и вызывают друг у друга методы.

[8]Исходный код PlantUML

2.2.3. Рисуемые объекты

Двумерный рисуемый объект (спрайт)

Класс `Sprite` описывает общие для всех рисуемых объектов в программе поведение и свойства. В графических фреймворках часто начало координат находится в верхнем левом или нижнем левом углу. Однако, очень часто, когда пишутся какие-то игры или другие приложения с использованием графики в качестве координат используется центр объекта. То есть необходимо условиться, что X и Y – это центр любого визуального объекта на канве. И, следовательно, удобно хранить не длину с шириной, а половину длины и половину ширины. А границы объекта, соответственно, будут отдаваться через геттеры и сеттеры. Дополнительно следует указать спрайту, что он умеет обновляться и рендериться, а его наследники уже смогут самостоятельно решать, как именно они хотят это делать.

Листинг 2.43: Абстрактный рисуемый объект – спрайт

```
package ru.gb.jdk.two.online;

import java.awt.*;

public abstract class Sprite {
    protected float x;
    protected float y;
    protected float halfWidth;
    protected float halfHeight;

    protected float getLeft() { return x - halfWidth; }
    protected void setLeft(float left) { x = left + halfWidth; }
    protected float getRight() { return x + halfWidth; }
    protected void setRight(float right) { x = right - halfWidth; }
    protected float getTop() { return y - halfHeight; }
    protected void setTop(float top) { y = top + halfHeight; }
    protected float getBottom() { return y + halfHeight; }
    protected void setBottom(float bottom) { y = bottom - halfHeight; }

    protected float getWidth() { return 2f * halfWidth; }
    protected float getHeight() { return 2f * halfHeight; }

    void update(MainCanvas canvas, float deltaTime) { }
    void render(MainCanvas canvas, Graphics g) { }
}
```

Конкретный рисуемый объект

Инстанцировать абстрактный класс нельзя, поэтому, нужно создать класс шарика, который будет перемещаться по экрану. В конструкторе шарику задаются случайные размеры с определённым разбросом. Чтобы не усложнять пример отдельным объектами, описывающими физику мира, непосредственно объекту шарика будут заданы скорости по осям X и Y, и цвет.

Листинг 2.44: Рисуемый объект

```
package ru.gb.jdk.two.online;

import java.awt.*;
import java.util.Random;

public class Ball extends Sprite {
    private static Random rnd = new Random();
    private final Color color;
    private float vX;
    private float vY;

    Ball() {
        halfHeight = 20 + (float) (Math.random() * 50f);
        halfWidth = halfHeight;
        color = new Color(rnd.nextInt());
        vX = 100f + (float) (Math.random() * 200f);
        vY = 100f + (float) (Math.random() * 200f);
    }

    @Override
    void update(MainCanvas canvas, float deltaTime) {
        x += vX * deltaTime;
        y += vY * deltaTime;

        if (getLeft() < canvas.getLeft()) {
            setLeft(canvas.getLeft());
            vX = -vX;
        }
        if (getRight() > canvas.getRight()) {
            setRight(canvas.getRight());
            vX = -vX;
        }
        if (getTop() < canvas.getTop()) {
            setTop(canvas.getTop());
            vY = -vY;
        }
        if (getBottom() > canvas.getBottom()) {
            setBottom(canvas.getBottom());
            vY = -vY;
        }
    }

    @Override
    void render(MainCanvas canvas, Graphics g) {
        g.setColor(color);
        g.fillOval((int) getLeft(), (int) getTop(),
              (int) getWidth(), (int) getHeight());
    }
}
```

В классе шарика переопределяются методы обновления и рендеринга. Самый простой рендер – объекту графики задаётся цвет текущего шарика и вызывается метод `fillOval()`, которому передаются левая и верхняя координаты, ширина и высота. Несмотря на то, что объекты содержат поля типа `float`, работа происходит с пиксельной системой координат, а значит необходимо переводить в целые числа[9].

В методе обновления к текущим координатам шарика прибавляется расстояние, которое должен был преодолеть шарик за то время пока канва спала и рендерилась.

$$ball(x_{new}, y_{new}) = ball(x + vx * \delta t, y + vy * \delta t).$$

Дополнительно, обрабатываются отскоки от границ панели, то есть описаны четыре условия, что при достижении границы меняется направление вектора.

В основном классе делается очень прямолинейно – создаётся массив из спрайтов, способный удержать десять шариков. В методе обновления каждый шарик из массива необходимо попросить обновиться, а в методе рендеринга – дать команду на отрисовку.

Реализация обновления и отрисовки остаётся самим объектам, то есть инкапсулируется в них. Только каждый объект сам по себе знает, как именно ему обновляться с течением времени, и как рисоваться, а основной экран управляет на более высоком уровне – на какой канве, когда и что рисовать.

В конструкторе добавляется простой цикл инициализирующий приложение десятью шариками.

Листинг 2.45: Управление объектами приложения

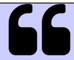
```
private final Sprite[] sprites = new Sprite[10];

private MainWindow() {
    // ...
    for (int i = 0; i < sprites.length; i++) {
        sprites[i] = new Ball();
    }
    // ...
}

private void update(MainCanvas canvas, double deltaTime) {
    for (int i = 0; i < sprites.length; i++) {
        sprites[i].update(canvas, deltaTime);
    }
}

private void render(MainCanvas canvas, Graphics g) {
    for (int i = 0; i < sprites.length; i++) {
        sprites[i].render(canvas, g);
    }
}
```

[9]Такой способ, конечно же, не подходит для реальных проектов, там необходимо всё сразу переводить в «мировые координаты» (например принять центр экрана за 0, верхний-левый угол за -1 нижний-правый за 1, как это делается в OpenGL) чтобы рендерить экраны.

Напомню, что самое главное, что необходимо понять из этого приложения – это взаимодействия и взаимовлияния объектов. Наследование, полиморфизм, инкапсуляция поведений и свойств.

2.2.4. Интерфейсы

Пример без интерфейсов



Начать разговор об интерфейсах я решил с создания отдельного класса фона, но сразу столкнулся с необходимостью думать головой...

На первый взгляд, логично было бы предположить, что фон – это спрайт, имеющий прямоугольную форму и всегда рисующийся первым. Но, есть затруднения, связанные с таким подходом: при изменении размеров окна фон тоже желательно изменить в размерах, а это лишние слушатели и десятки строк кода, поэтому при отрисовке объекта фона гораздо проще будет дать команду канве на изменение фона.

Листинг 2.46: Фон, как наследник спрайта и изменения в основном классе

```java
package ru.gb.jdk.two.online;

import java.awt.*;

public class Background extends Sprite {
    private float time;
    private static final float AMPLITUDE = 255f / 2f;
    private Color color;

    @Override
    public void update(MainCanvas canvas, float deltaTime) {
        time += deltaTime;
        int red = Math.round(AMPLITUDE + AMPLITUDE * (float) Math.sin(time * 2f));
        int green = Math.round(AMPLITUDE + AMPLITUDE * (float) Math.sin(time *
            3f));
        int blue = Math.round(AMPLITUDE + AMPLITUDE * (float) Math.sin(time));
        color = new Color(red, green, blue);
    }

    @Override
    public void render(MainCanvas canvas, Graphics g) {
        canvas.setBackground(color);
    }
}

// MainCanvas
sprites[0] = new Background();
for (int i = 1; i < sprites.length; i++) {
    sprites[i] = new Ball();
}
```

Цвет фона меняется синусоидально по каждому из трёх компонентов цвета, поэтому изменение происходит плавно (рис. 2.11).

Для реализации фона от спрайта, фактически, нужно только поведение, а свойства не нужны. Но и отказываться от наследования не очень правильно, потому что тогда не получится фон единообразно в составе массива спрайтов обновлять. Эти факты напрямую намекают на унификацию поведения – на интерфейс.

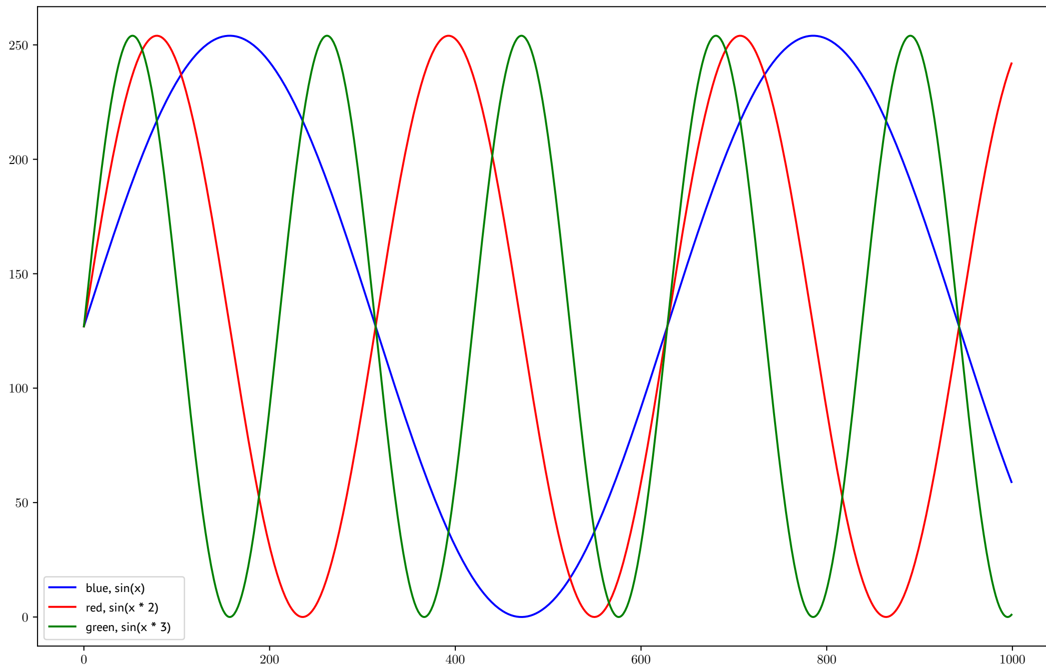
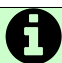


Рис. 2.11: График изменения значений компонентов цвета

Понятие интерфейса

 Механизм наследования очень удобен, но он имеет свои ограничения. В частности, в языке Java допустимо наследование только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java проблему отсутствия множественного наследования частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. Один класс может применить к себе множество интерфейсов. Правильно говорить «реализовать интерфейс». Интерфейс можно очень примерно представить как очень абстрактный класс. Интерфейс – это описание методов.

Интерфейс – это описание способов взаимодействия с объектом. Интерфейсы определяют функционал, не имеющий конкретной реализации.

Примером интерфейса в реальной жизни может быть интерфейс управления автомобилем, интерфейс взаимодействия с компьютером или интерфейс USB, так, компьютеру не важно, что именно находится по ту сторону провода – флеш накопитель, веб-камера или мобильный телефон, а важно, что компьютер умеет работать с интерфейсом USB, отправлять туда байты или получать. Потоки ввода-вывода, которые были изучены – это тоже своего рода интерфейс, соединяющий не важно какого отправителя, например, программный код и не важно какого получателя, например, файл.

Интерфейсы объявляются также, как классы, и могут иметь очень похожую на класс структуру, то есть быть вложенным или внутренним. Чаще всего каждый отдельный интерфейс описывают в отдельном файле, также как класс, но используя ключевое слово `interface`. Ниже показаны примеры интерфейсов, человек и бык, в которых описаны методы «ходить» и «издавать звуки».

Все методы во всех интерфейсах всегда публичные, и в классическом варианте (до Java 1.8) не имеют никакой реализации. Поскольку все методы всегда публичные, то этот модификатор принято не писать.

Листинг 2.47: Примеры интерфейсов

```
package ru.gb.jdk.two.online.samples;

public interface Human {
    public void walk();
    public void talk();
}

package ru.gb.jdk.two.online.samples;

public interface Bull {
    void walk();
    void talk();
}
```

Реализация интерфейса

Продолжая учебный пример: созданы классы мужчина и бык. Класс мужчины реализовывает интерфейс человека, а класс быка – быка.

Для реализации интерфейса необходимо переопределить все его методы, либо сделать класс абстрактным.

Множественного наследования нет, но существует возможность реализовать любое количество интерфейсов.

Листинг 2.48: Классы, реализующие интерфейс

```
package ru.gb.jdk.two.online.samples;

public class Man implements Human {
    @Override
    public void walk() {
```

```
        System.out.println("Walks on two feet");
    }

    @Override
    public void talk() {
        System.out.println("Talks meaningful words");
    }
}

package ru.gb.jdk.two.online.samples;

public class Ox implements Bull {
    @Override
    public void walk() {
        System.out.println("Walks on hooves");
    }

    @Override
    public void talk() {
        System.out.println("MooOooOooo00oo");
    }
}
```

Одним из самых удобных следствий применения интерфейсов является возможность объявлять не только классы и создавать объекты, но и создать идентификторы, которые ссылаются на объект, реализующий интерфейс. То есть, по идентификатору типа интерфейса могут лежать абсолютно не связанные между собой объекты, главное, чтобы они реализовывали интерфейс. При этом сохраняется возможность работать с методами интерфейса которые могут быть для разных классов по-разному реализованы. Это иная форма изученного ранее **полиморфизма**.

Листинг 2.49: Интерфейсные переменные

```
package ru.gb.jdk.two.online.samples;

public class Main {
    public static void main(String[] args) {
        Man man0 = new Man(); //class Man
        Ox ox0 = new Ox(); // class Ox
        Human man1 = new Man(); // interface Human
        Bull ox2 = new Ox(); // interface Bull
    }
}
```

Для демонстрации ещё одного способа применения интерфейсов, будет описан класс минотавра[10], реализовывающий интерфейсы человека и быка своим собственным способом, а именно, ходил на ногах человека, но не мычал, как бык, а загадывал загадки.

При использовании интерфейсов важно то, что классы не связаны между собой наследованием, а обращение к ним единообразно.

[10] мифический персонаж с телом человека и головой быка.

Интересно то, что в программе таким образом появляется возможность обратиться к минотавру не только как к человеку, но и как к быку, то есть гипотетически, можно создать некоторого Тесея, управляющего большим количеством минотавров.

Листинг 2.50: Реализация множества интерфейсов

```java
package ru.gb.jdk.two.online.samples;

public class Main {
   private static class Minotaurus implements Human, Bull {
      @Override public void walk() {
         System.out.println("Walks on two legs");
      }

      @Override public void talk() {
         System.out.println("Asks you a riddle");
      }
   }
   public static void main(String[] args) {
      Bull minos0 = new Minotaurus();
      Human minos1 = new Minotaurus();
      Minotaurus minos = new Minotaurus();
      Human man1 = new Man();
      Bull ox2 = new Ox();
      Bull[] allBulls = {ox2, minos0, minos};
      Human[] allHumans = {man1, minos, minos1};
   }
}
```

Также важно, что в интерфейсах разрешено наследование. То есть, один интерфейс может наследоваться от другого интерфейса, соответственно, при реализации такого, наследующего интерфейса, необходимо переопределять не только методы интерфейса, но и методы всех его родителей, также, как если бы происходило переопределение методов абстрактного класса.

Следует обратить особенное внимание на то, что в интерфейсах разрешено множественное наследование.

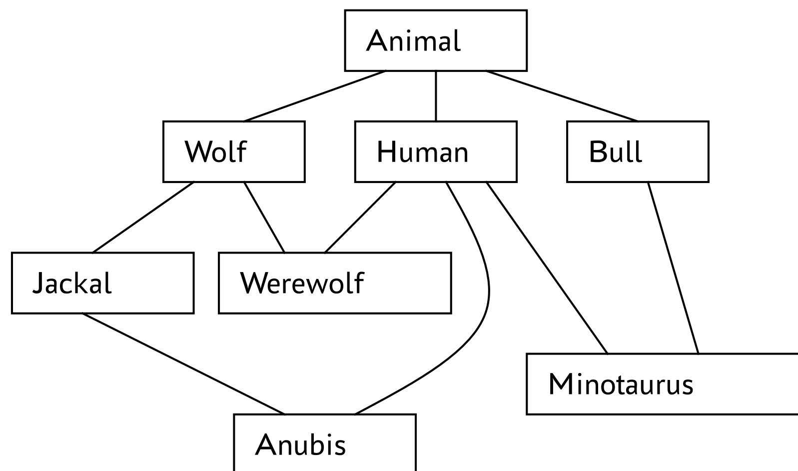


Рис. 2.12: Наследование интерфейсов

Вопросы для самопроверки

1. Программный интерфейс – это:
   (a) окно приложения в ОС;
   (b) реализация методов объекта;
   (c) объявление методов, реализуемых в классах.
2. Интерфейсы нужны для:
   (a) компенсации отсутствия множественного наследования;
   (b) отделения API и реализации;
   (c) оба варианта верны.
3. Интерфейсы позволяют:
   (a) удобно создавать новые объекты, не связанные наследованием;
   (b) единообразно обращаться к методам объектов, не связанных наследованием;
   (c) полностью заменить наследование.

Применение интерфейса

В описанном ранее примере интерфейс помогает решить проблему единообразия поведения спрайтов и фона, при их различии в свойствах. То есть, сложилась ситуация в которой существует необходимость хранить в одном массиве объекты со схожим поведением, но наследовать их друг от друга не совсем логично.

Листинг 2.51: Интерфейс обновляемого и рисуемого объекта

```
package ru.gb.jdk.two.online;

import java.awt.*;

public interface Interactable {
    void update(MainCanvas canvas, float deltaTime);
    void render(MainCanvas canvas, Graphics g);
}
```

В качестве решения описан интерфейс `Interactable`, содержащий методы обновления и рендеринга без реализации.

> ⚠ Если описывать ещё более гибкое приложение, нужно создавать два интерфейса, `Updatable` и `Renderable`, чтобы иметь возможность отделить рисуемые объекты от обновляемых.

В данном случае интерфейс описывает объекты, которые должны уметь рисоваться и обновляться. В спрайте и фоне интерфейс реализуется. При этом получается, то фон никак не связан со спрайтом, но при этом, оба умеют рисоваться и обновляться, благодаря интерфейсу. Далее, при смене массива спрайтов на массив интерактивностей приложение не сломается.

Листинг 2.52: Реализация интерфейса объектами приложения

```
public abstract class Sprite implements Interactable

public abstract class Background implements Interactable
```

Создание библиотечных классов

Если обратить внимание на развитие повествования по курсу, можно заметить, что сначала произошёл выход за пределы одного метода (методы, помимо `main`), потом за пределы одного класса (классы котиков, собачек, и т.д.), затем за пределы одного пакета (логическое разделение классов), за пределы программного кода (потоки ввода-вывода), а теперь код, который возможно использовать несколькими программами.

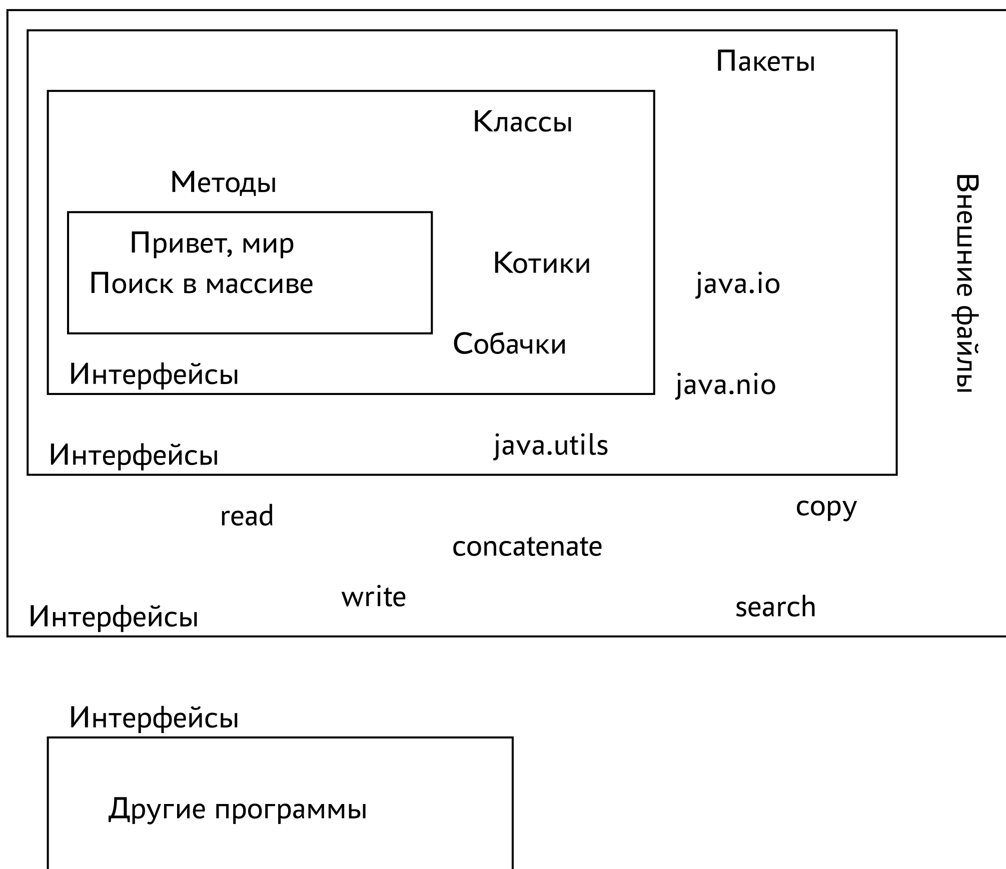
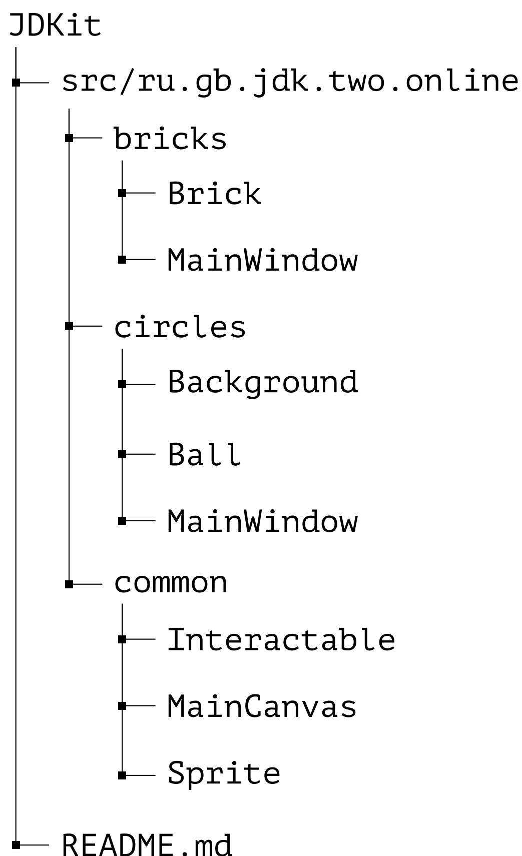
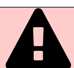


Рис. 2.13: Применение интерфейсов для отделения единиц компиляции

У классов канвы и спрайта, а также у интерфейса, нет никакой специфики, они ничего не «знают» о том, какие объекты существуют в программе и как эти объекты взаимодействуют между собой. Эти классы и интерфейс применимы, по сути, где угодно, не только в этой конкретной программе с этими конкретными классами. С использованием таких общих фрагментов кода становится возможным достаточно быстро написать вторую игру: новый пакет, новый класс, скопированный код от основного окна шариков. А полностью копировать спрайты и интерфейсы не целесообразно. Сделав правильное дробление по пакетам становится очевидно, что существует общий библиотечный пакет, и какие-то приложения с конкретными реализациями.



В общий пакет классы скопировались без проблем, шарики перенеслись с минимальными изменениями – только публичные модификаторы понадобились. В главном окне с будущими летающими квадратиками создан аналогичный конструктор, размеры, положение. Но общей канвой воспользоваться невозможно. Это происходит, потому что канва может принимать в качестве параметра в конструкторе только основное окно из пакета кружочков. И далее канва уже у класса с кружочками вызывает метод `onDrawFrame()`.



Привязка к классу ограничивает возможности.

Решение на поверхности – использование интерфейсов. Необходимо написать интерфейс, который может по смыслу называться, например, `CanvasRepaintListener`, который будет уметь ожидать от канвы вызов метода и реализовывать его.

Листинг 2.53: Интерфейс слушателя событий канвы

```
1 package ru.gb.jdk.two.online.common;
```

```
2
3 import java.awt.*;
4
5 public interface CanvasRepaintListener {
6     void onDrawFrame(MainCanvas canvas, Graphics g, float deltaTime);
7 }
```

Такой интерфейс логично создавать в общем пакете и переписать канву так, чтобы она принимала на вход не класс, а объект, реализующий интерфейс. Далее, оба слушателя реализуются через интерфейс.

Листинг 2.54: Использование интерфейса на канве

```
1 public class MainCanvas extends JPanel {
2     private final CanvasRepaintListener controller;
3     private long lastFrameTime;
4
5     public MainCanvas(CanvasRepaintListener controller) {
6         this.controller = controller;
7         lastFrameTime = System.nanoTime();
8     }
```

Интерфейс может быть реализован классом, а окна приложений – это тоже классы. Это позволяет не только наследоваться от классов фреймворка Swing, но и реализовывать интерфейсы, описанные внутри приложения. Следовательно, окно без изменений продолжает передавать себя в конструктор, а метод интерфейса уже реализован. Чтобы подчеркнуть, что это реализация интерфейса – дописана аннотация `@Override`.

Листинг 2.55: Интерфейс обновляемого и рисуемого объекта

```
1 public class MainWindow extends JFrame implements CanvasRepaintListener {
2     private MainWindow() { ... }
3
4     @Override
5     public void onDrawFrame(MainCanvas canvas, Graphics g, float deltaTime) {
          ... }
```

Особенности интерфейсов

Интерфейсы были значительно переработаны в Java 1.8, было добавлено довольно много механизмов, об одном из которых нельзя не сказать. Реализация интерфейсов по умолчанию. Пример будет построен на основе тех интерфейсов, которые уже написаны – человек и бык. Очевидно, что именно у этих интерфейсов возможны реализации по умолчанию, например, для действия «ходить»: человек ходит на двух ногах, а бык на четырёх копытах. Для описания реализации по умолчанию используется ключевое слово `default`. Если написать реализацию, но не использовать данное ключевое слово, произойдёт ошибка компиляции (или среда разработки укажет на ошибочность такой конструкции).

Листинг 2.56: Реализация по умолчанию

```
1 package ru.gb.jdk.two.online.samples;
2
```

```
3  public interface Human {
4     default void walk() {
5        System.out.println("Walks on two feet");
6     }
7  
8     public void talk();
9  }
10 
11 package ru.gb.jdk.two.online.samples;
12 
13 public interface Bull {
14    void walk() { // compile time error
15       System.out.println("Walks on four hooves");
16    }
17    void talk();
18 }
```

Первое, и самое очевидное следствие использования реализации по умолчанию – отсутствие необходимости переопределять все методы в классах, реализующих эти интерфейсы, что делает интерфейс, в свою очередь, чуть более похожим на класс. Реализованные по умолчанию интерфейсы могут задействовать созданные в этом интерфейсе поля, а наличие в интерфейсе полей делает его ещё более похожим на класс. Все поля в интерфейсах статические и неизменяемые, а если заменить публичный модификатор доступа на другой – будет ошибка компиляции.

Листинг 2.57: Использование полей в интерфейсах

```
1  public interface Bull {
2     public static final int amount = 2;
3     default void walk() {
4        System.out.println("Walks on " + amount + " hooves");
5     }
6     void talk();
7  }
```

2.2.5. Анонимные классы

Понятие и применение

Программные интерфейсы открывают перед разработчиком широчайшие возможности по написанию более выразительного кода. Одна из наиболее часто используемых возможностей – анонимные классы.

Класс – это новый тип данных для программы. Классы бывают вложенными и внутренними. Внутренние классы – это классы, которые пишутся внутри других классов, которые в свою очередь описаны в файле. А также вложенные или локальные классы, которые возможно объявлять непосредственно в методах, и работать с ними, как с обычными классами. Анонимный класс, что довольно очевидно - это класс без названия. Далее приводится пример создания интерфейса `MouseListener` и описания в нём методов `mouseUp()`, `mouseDown()`. В основной части программы описан класс, реализующий этот интерфейс, то есть переопределяющий все его методы. Далее в методе `main()` создаётся экземпляр этого класса и появляется возможность использовать его методы.

Листинг 2.58: Способ использования API через именованный класс

```
public interface MouseListener {
    void mouseUp();
    void mouseDown();
}

private static class MouseAdapter implements MouseListener {
    @Override public void mouseUp() { }
    @Override public void mouseDown() { }
}

public static void main(String[] args) {
    MouseAdapter m = new MouseAdapter();
    m.mouseDown();
    m.mouseUp();
}
```

Очень часто, элементы управления (кнопки, события входящих датчиков (клавиатура, мышка), сеть требуют на вход каких-то обработчиков собственных данных, которые будут слушать конкретный источник данных, отлавливать события и знать что делать. Это делается через интерфейсы. С точки зрения программы, создаётся некий метод, например, метод добавления к кнопке слушателя. Далее, если в элемент управления в качестве слушателя передаётся какой-то объект, реализующий нужный интерфейс, то этот объект начнёт ловить события и как-то их обрабатывать.

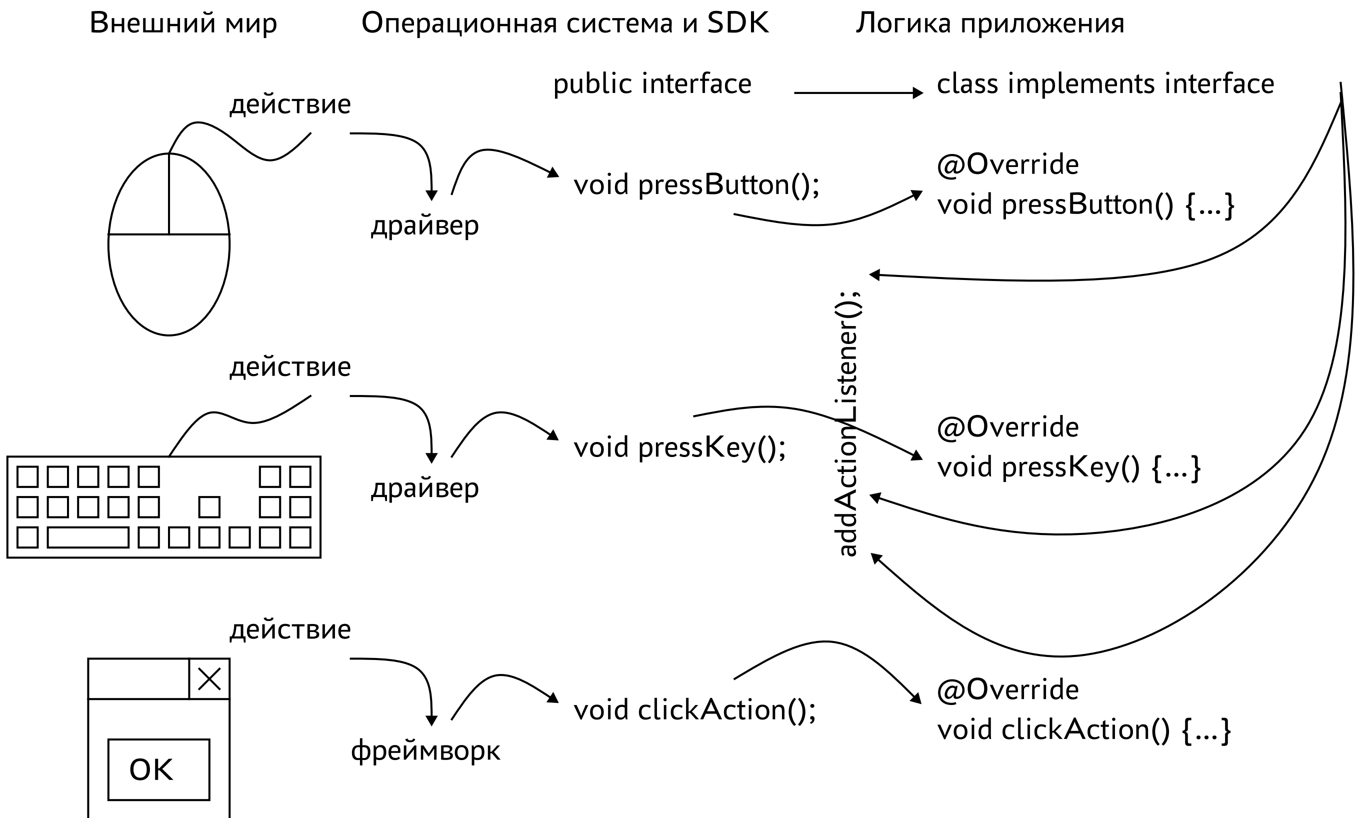


Рис. 2.14: Применение слушателей событий

Для полноценного примера (листинг 2.59) следует описать метод, принимающий на вход

`MouseListener` (строка 1). И передать в метод объект (строка 13), предполагая, что внутри объекта для данного конкретного случая описано, как именно должна вести себя программа, когда кто-то нажал или отпустил кнопку мышки.

Допустим, есть параметр метода `MouseListener m` и необходимость создать туда некоторый экземпляр, который реализует этот интерфейс. Но есть класс, который это делает: `MouseAdapter`. Проще будет создать экземпляр адаптера, но без идентификатора (строка 14).

Часто такие классы создаются не просто без названия экземпляра, но и вовсе без имени, прямо в аргументе методов. Действительно, зачем классу имя, если он будет использован только один раз и только в этом методе для создания одного единственного объекта? Класс `MouseAdapter` идеально выполняет критерий **S** из принципов **SOLID** – Single Responsibility, делает только одно полезное дело – реализует интерфейс `MouseListener`. Но раз дело только одно – можно его выполнять и без размышлений о названии класса. Для создания интерфейсной переменной есть немного необычный синтаксис, на 15 строке. Получается что создаётся один экземпляр анонимного класса, который реализует интерфейс `MouseListener`. И созданный здесь же экземпляр данного класса кладётся в идентификатор.

Возможно также не создавать интерфейсный идентификатор, а сразу передать реализующий экземпляр в аргумент метода. Получится, что в метод передаётся новый экземпляр анонимного класса который реализует интерфейс слушателя, и тут же даётся описание этого класса, в котором переопределяются соответствующие методы (строка 20).

Ещё раз: **анонимные классы** – это классы, не имеющие названия и реализующие какой-то интерфейс.

Листинг 2.59: Возможные способы создания обработчиков событий

```java
private static void addMouseListener(MouseListener l) {
    l.mouseDown();
    l.mouseUp();
}

private static class MouseAdapter implements MouseListener {
    @Override public void mouseUp() { }
    @Override public void mouseDown() { }
}

public static void main(String[] args) {
    MouseAdapter m = new MouseAdapter();
    addMouseListener(m);
    addMouseListener(new MouseAdapter());
    MouseListener l = new MouseListener() {
        @Override public void mouseUp() { }
        @Override public void mouseDown() { }
    };
    addMouseListener(l);
    addMouseListener(new MouseListener() {
        @Override public void mouseUp() { }
        @Override public void mouseDown() { }
});
```

Альтернативы

Существует возможность избежать использования анонимных классов и реализации сложных интерфейсов. Например, использовать адаптеры.

Для панели на которой рисовались летающие шарики (2.2.2) существует слушатель и метод добавления реализации слушателя мышки, очень похожий на созданный в учебных целях.

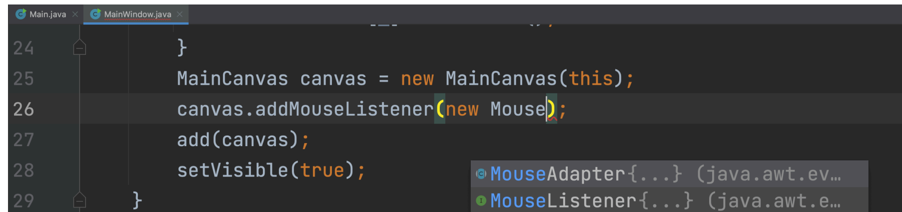


Рис. 2.15: Метод добавления слушателя мышки

Если в аргументе этого метода начать писать `new`, среда разработки предложит реализовать интерфейс `MouseListener`, но также предложит ещё один вариант – `MouseAdapter`. В исходниках класса `MouseAdapter` видно, что это класс, реализующий несколько интерфейсов, но только формально, то есть все реализации пустые.

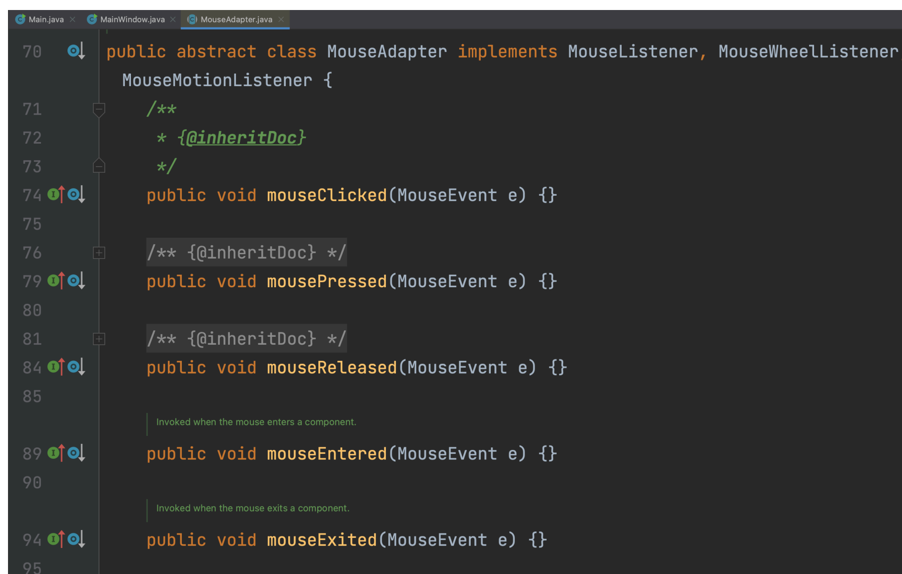


Рис. 2.16: Содержимое класса `MouseAdapter`

Это позволяет переопределять не все, а только некоторые методы интерфейса, значительно экономя место в коде приложения, если, например, нужна реакция только на одно какое-то действие.

```
26        }
27        MainCanvas canvas = new MainCanvas(this);
28        canvas.addMouseListener(new MouseAdapter() {
29            @Override
30            public void mouseReleased(MouseEvent e) {
31                super.mouseReleased(e);
32            }
33        });
```

Рис. 2.17: Частичная реализация интерфейса

> Если попытаться это корректно перевести на русский язык, должно получиться: создай новый экземпляр анонимного класса, который наследуется от класса `MouseAdapter`, реализующего нужный интерфейс и переопредели этот конкретный метод. Остальные оставь пустыми, потому что остальные действия можно игнорировать.

Как избежать таких «многоэтажных» и многострочных конструкций и при этом получить понятный код? Реализовать интерфейс в том классе, в котором в данный момент пишется код, и переопределить все методы интерфейса. В требуемые методы – написать реализацию, а туда, где требуется объект, реализующий интерфейс – передать ссылку `this`.

Листинг 2.60: Пример реализации интерфейса «собой»

```
1  public class MainWindow extends JFrame implements
2        CanvasRepaintListener, MouseListener {
3     MainWindow() {
4        // ...
5        canvas.addMouseListener(this);
6     }
7     @Override public void mouseClicked(MouseEvent e) { }
8     @Override public void mousePressed(MouseEvent e) { }
9     @Override public void mouseReleased(MouseEvent e) {
10       System.out.println("Clicked!");
11    }
12    @Override public void mouseEntered(MouseEvent e) { }
13    @Override public void mouseExited(MouseEvent e) { }
14 }
```

SOLID

Буква	Аббревиатура	Пояснение
S	SRP	Принцип единственной ответственности (single responsibility principle). Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.
O	OCP	Принцип открытости/закрытости (open-closed principle). «Программные сущности ... должны быть открыты для расширения, но закрыты для модификации».
L	LSP	Принцип подстановки Лисков (Liskov substitution principle). «Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».
I	ISP	Принцип разделения интерфейса (interface segregation principle). «Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»
D	DIP	Принцип инверсии зависимостей (dependency inversion principle). «Зависимость на Абстракциях. Нет зависимости на что-то конкретное»

Вопросы для самопроверки

1. Программный интерфейс — это способ
   (a) рисования объектов;
   (b) взаимодействия объектов;
   (c) взаимодействия программы с пользователем.
2. Анонимный класс — это класс без
   (a) интерфейса;
   (b) объекта;
   (c) имени.
3. Поле в интерфейсе
   (a) невозможно;
   (b) public static final;
   (c) private final.
4. Метод по-умолчанию
   (a) можно переопределять;
   (b) можно не переопределять;
   (c) можно использовать с полем интерфейса;
   (d) все варианты верны.

2.2.6. Исключения в графических интерфейсах пользователя

Поскольку графический интерфейс пользователя - это всегда многопоточность, и привычного терминала под рукой чаще всего нет, то возникают особенности обработки исключений. Как ловить? Как показывать?

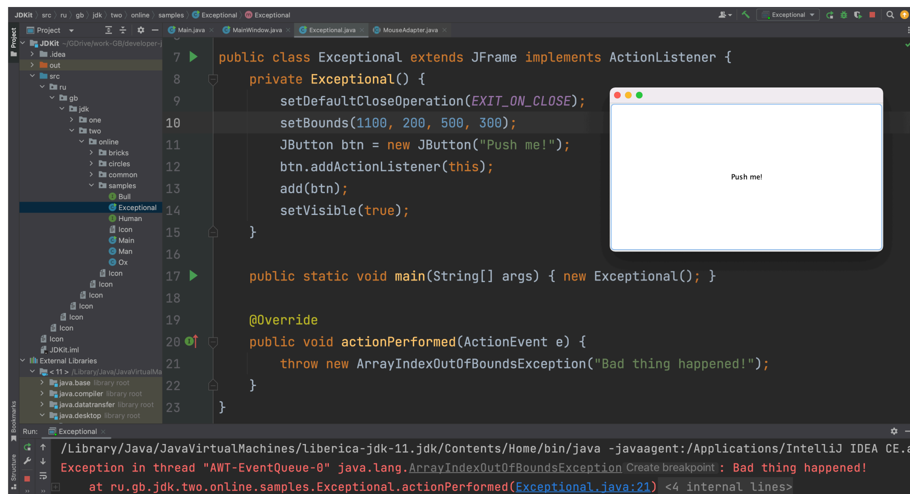


Рис. 2.18: Исключение, которое никогда не увидят

Например, есть окно, на котором есть кнопка. У кнопки есть обработчик, в котором что-то идёт не по плану, например, выход за пределы массива при подсчёте. Достаточно типичная ситуация. Возникает законный вопрос - как такое исключение поймать?

Листинг 2.61: Пример окна с исключением

```java
package ru.gb.jdk.two.online.samples;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Exceptional extends JFrame implements ActionListener {
    private Exceptional() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setBounds(1100, 200, 500, 300);
        JButton btn = new JButton("Push me!");
        btn.addActionListener(this);
        add(btn);
        setVisible(true);
    }

    public static void main(String[] args) { new Exceptional(); }

    @Override
    public void actionPerformed(ActionEvent e) {
        throw new ArrayIndexOutOfBoundsException("Bad thing happened!");
    }
}
```

Если бы такое же исключение было выброшено в консольном приложении, программа бы завершилась, здесь же видно, что, несмотря на исключение в консоли, окно всё ещё открыто и приложение работает. Ясно видно, что исключения не завершают приложение.

Во-первых, следует применять правильный способ создания главных окон в Swing, эта конструкция уже не должна быть магической: у класса `SwingUtilities` есть статический метод,

в который передаётся экземпляр анонимного класса, реализующего интерфейс, и в переопределяемом методе создаётся окно.

Листинг 2.62: Верный способ создавать окна в Swing

```
public static void main(String[] args) {
   SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
         new Exceptional();
      }
   });
}
```

Исключение происходит в специальном потоке EDT – Event Dispatching Thread. Этот поток совершает диспетчеризацию всех событий, происходящих во фреймворке Swing и является генератором других потоков. Метод из листинга 2.62 явно создаёт `JFrame` именно под управлением EDT.

Если внимательно изучить текст исключения внизу экрана на рисунке 2.18, очевидно, что исключение возникло в потоке с названием `AWT-EventQueue-0`. Наличие у потока номера говорит о том, что таких очередей событий у приложения может быть много, и в каких-то из них могут возникать исключения.

Исключение происходит в потоке. Обработчик исключений тоже содержится в потоке и называется `Thread.UncaughtExceptionHandler` и является интерфейсом. Интерфейс содержит один метод – непойманное исключение, который принимает на вход поток, в котором произошло исключение и объект исключения, которое произошло.

Листинг 2.63: Пример окна с исключением

```
public class Exceptional extends JFrame implements
      ActionListener, Thread.UncaughtExceptionHandler {

   @Override
   public void uncaughtException(Thread t, Throwable e) {

   }
}
```

Такие обработчики уже написаны и встроены в среду исполнения Java, но они только пишут в консоль стектрейс. Переопределив метод интерфейса в приложении – появляется возможность реагировать на исключения более сложно.

Листинг 2.64: Пример окна с исключением

```
private Exceptional() {
   Thread.setDefaultUncaughtExceptionHandler(this);
   //...
}

@Override
public void uncaughtException(Thread t, Throwable e) {
   JOptionPane.showMessageDialog(
         null, e.getMessage(),
```

```
10            "Exception!", JOptionPane.ERROR_MESSAGE);
11  }
```

В конструкторе (листинг 2.64), на строке 2 для потока устанавливается обработчик исключений по-умолчанию, передаётся собственный объект окна. В самой функции обработки (строка 6), например, выводится на экран модальное окно с текстом исключения. Запустив приложение видно, что в консоли среды разработки нет сообщений об исключениях. Благодаря программным интерфейсам.

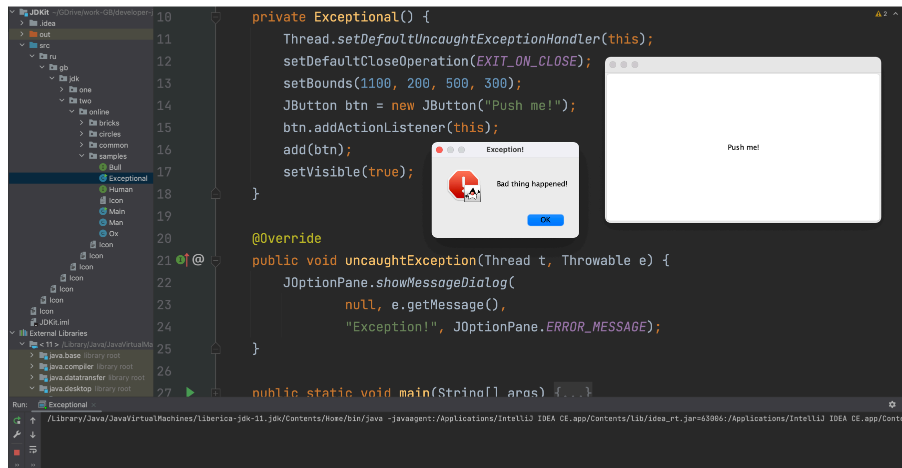


Рис. 2.19: Результат обработки исключения

Практическое задание

1. Полностью разобраться с кодом.
2. Для приложения с шариками описать появление и убирание шариков по клику мышки левой и правой кнопкой соответственно.
3. Написать, выбросить и обработать такое исключение, которое не позволит создавать более, чем 15 шариков.
4. ** Написать ещё одно приложение, в котором на белом фоне будут перемещаться изображения формата png, лежащие в папке проекта.

Термины, определения и сокращения

`finally` — часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.

`throws` — ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.

`throw` — оператор, активирующий (выбрасывающий) объект исключения.

`try...catch` — двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.

BIOS — (англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.

CI — (англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CLI — (англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».

cp1251 — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK — (от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.

Stacktrace — часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.

String — Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String

Swing — библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.

Typecasting — преобразование типов переменных в типизированных языках программирования

UTF-8 — (от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.

Асинхронность — вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.

Ввод-вывод — взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации

Вложенный класс — статический класс, объявленный внутри другого класса.

Внутренний класс — нестатический класс, объявленный внутри другого класса.

Загрузчик — системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.

Идентификатор — идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

2.3. Инструментарий: Обобщения

В предыдущем разделе

- программные интерфейсы — понятие и принцип работы;
- ключевое слово implements;
- Наследование и множественное наследование интерфейсов;
- реализация, реализации по-умолчанию;
- частичная реализация интерфейсов, адаптеры;
- анонимные классы.

В этом разделе

Смысл обобщений в программировании примерно такой же, как и в обычной разговорной речи – отсутствие деталей реализации. С точки зрения проектирования обобщения – это более сложный полиморфизм. Будет рассмотрены diamond operator, обобщённые методы и подстановочные символы (Wildcards). Применение обобщений для ограничений типов сверху и снизу. Выведение и стирание типов, целевые типы и загрязнение кучи.

- Обобщения;
- Wildcard;
- Diamond operation;
- Стирание типа;
- Выведение типа;
- Сырой (raw) тип;
- Целевой тип.

2.3.1. Понятие обобщения

Работа без обобщений

Обобщения – это механизм создания общего подхода к реализации одинаковых алгоритмов, но с разными данными. Обобщения – это некоторые конструкции, позволяющие работать с данными не задумываясь о том, какие именно данные лежат внутри структуры или метода (строки, числа или коты). Обобщения позволяют единообразно работать с разными типами данных.

Обобщённое программирование в Java позволяет создавать классы, интерфейсы и методы, которые могут работать с различными типами данных. В результате, код становится более универсальным и повторно используемым.

Для приведения примера поведения контейнера, который может хранить данные любого типа необходимо создать некоторый класс, который будет хранить внутри `Object` (любые данные Java). На примере чисел, в основном методе приложения созданы два экземпляра коробок с числами. Числа, которые сохранены в коробке желательно иметь возможность не только хранить, но и использовать, например, производить операцию сложения.

Листинг 2.65: Контейнер, хранящий любые данные

```
1 private static class Box {
```

```
2       private Object obj;
3       public Box(Object obj) {
4          this.obj = obj;
5       }
6
7       public Object getObj() {
8          return obj;
9       }
10      public void setObj(Object obj) {
11         this.obj = obj;
12      }
13      public void printInfo() {
14         System.out.printf("Box (%s): %s\n",
15               obj.getClass().getSimpleName(),
16               obj.toString());
17      }
18   }
19   public static void main(String[] args) {
20      Box b1 = new Box(20);
21      Box b2 = new Box(30);
22      System.out.println(b1.getObj() + b2.getObj());
23   }
```

Поскольку в коробке, с точки зрения языка, хранятся объекты, а оператор сложения для объектов не определён, следует применить операцию приведения типов. Средства языка не запрещают создать больше коробок, например, со строками, и будет производиться уже не складывание чисел, а конкатенация строк.

Проблема такого способа в том, что при каждом получении данных из коробки необходимо делать приведение типов, чтобы указать какие именно в контейнере лежат данные. Данную проблему для программы возможно решить организационно, если, например, называть переменные в венгерской нотации[11].

Листинг 2.66: Согласование типов

```
1  public static void main(String[] args) {
2        Box b1 = new Box(20);
3        Box b2 = new Box(30);
4        System.out.println((Integer) b1.getObj() + (Integer) b2.getObj());
5
6        Box b3 = new Box("Hello, ");
7        Box b4 = new Box("World!");
8        System.out.println((String) b3.getObj() + (String) b4.getObj());
9  }
```

Вторая проблема в том, что данные в контейнере ничем не защищены. Java не запрещает менять данные внутри контейнера, поскольку для платформы это объект. Проблему возможно решить сделав проверку `instanceof` перед приведением типов.

Третья проблема в том, что все проблемы подобного рода проявляют себя только во время

[11] Венгерская нотация в программировании – соглашение об именовании переменных, констант и прочих идентификаторов в коде программ. Тип переменной указывается строчной буквой перед именем переменной, например для типа `int` переменная может называться `iMyVariable`.

исполнения приложения, то есть у конечного пользователя перед глазами, когда разработчик ничего исправить не может.

Листинг 2.67: Изменение типа хранения во время исполнения

```
public static void main(String[] args) {
   Box iBox1 = new Box(20);
   Box iBox2 = new Box(30);
   if (iBox1.getObj() instanceof Integer && iBox2.getObj() instanceof Integer) {
      int sum = (Integer) iBox1.getObj() + (Integer) iBox2.getObj();
      System.out.println("sum = " + sum);
   } else {
      System.out.println("The contents of the boxes differ by type");
   }
   iBox1.setObj("sdf"); // Java: "What can go wrong here? You can do it!"
}
```

Таким образом, в языке Java возможно создавать классы, которые могут работать с любыми типами данных, но при любом обращении к таким классам и данным необходимо делать достаточно сложные проверки.

Создание обобщения

- Java generics – это механизм языка, который позволяет создавать обобщенные (шаблонизированные) типы и методы в Java (особый подход к описанию данных и алгоритмов, позволяющий работать с различными типами данных без изменения внешнего описания);
- Java generics были добавлены в Java 1.5 и стали одной из наиболее важных новых функций в языке;
- Java generics работают только со ссылочными типами данных;
- Java generics предоставляют безопасность типов во время компиляции, что означает, что ошибки связанные с типами данных могут быть обнаружены на этапе компиляции, а не во время выполнения программы.

Для описания обобщения в треугольных скобках пишется буква <T>, чтобы обозначить Type, Тип. На этапе описания класса невозможно сказать, какого типа данные будут лежать в переменной во время исполнения (число, строка или кот).

Если написать T не в треугольных скобках при описании класса, то Java будет искать реально существующий класс, который она не видит.

Таким образом указывается, что это обобщение и тип будет задаваться при создании объекта. Естественно поменять его будет нельзя, потому что Java – это язык сильной статической типизации.

Листинг 2.68: Создание обобщённого контейнера

```
private static class GBox<T> {
   private T value;

   public GBox(T value) {
      this.value = value;
   }
```

```
7
8      public T getValue() {
9          return value;
10     }
11     public void setValue(T value) {
12         this.value = value;
13     }
14     public void showType() {
15         System.out.printf("Type is %s, with value %s\\n",
16                 value.getClass().getName(), getValue());
17     }
18 }
19
20 public static void main(String[] args) {
21     GBox<String> stringBox = new GBox<>("Hello!");
22     stringBox.showType();
23     GBox<Integer> integerBox = new GBox<>(12);
24     integerBox.showType();
25 }
```

При вызове конструктора такого объекта, будет указано, что конструктор ожидает указанный ранее тип, а не `T`. При указании типа в левой части, этот тип подставляется во все места класса, где компилятор обнаружит `T`. При получении значений из `integerBox` и `stringBox` не требуется преобразование типов, `integerBox.getValue()` сразу возвращает `Integer`, а `stringBox.getValue()` – `String`.

Если объект создан как `Integer`, то становится невозможно записать в него строку. При попытке написать такую строку кода, получится ошибка на этапе компиляции, то есть обобщения отслеживают корректность используемых типов данных.

По соглашению, переменные типа именуются одной буквой в верхнем регистре. Если обобщённых переменных более одной – они пишутся через запятую.

E – элемент (Element, Entity обширно используется Java Collections);
K – Ключ;
N – Число;
T – Тип;
V – Значение;
S, U, и т. п. — 2-й, 3-й, 4-й типы.

Ограничения обобщений

Обобщения накладывают на работу с собой некоторые ограничения.

1. Невозможно внутри метода обобщённого класса создать экземпляр параметризующего класса `T`, потому что на этапе компиляции об этом классе ничего не известно. Это ограничение возможно обойти, используя паттерн проектирования *абстрактная фабрика*.
2. Нельзя создавать внутри обобщения массив из обобщённого типа данных. Но всегда можно подать такой массив снаружи.
3. По причине отсутствия информации о параметризуещем классе, невозможно создать статическое поле типа. Конкретный тип для параметра `T` становится известен только при

созданий **объекта** обобщённого класса.
4. Нельзя создать исключение обобщённого типа.

Работа с обобщёнными объектами

При обращении к обобщённому классу необходимо заменить параметры типа на конкретные классы или интерфейсы, например строку, целое число или кота.

«Параметр типа» и «аргумент типа» – это два разных понятия. Когда объявляется обобщённый тип `GBox<T>`, то `T` является параметром типа, а когда происходит обращение к обобщённому типу, передается аргумент типа, например `Integer`.

Как и любое другое объявление переменной запись вида `GBox<Integer> integerBox` сам по себе не создаёт экземпляр класса `GBox`. Такой код объявляет идентифиактор типа `GBox`, но сразу уточняет, что это будет коробка с целыми числами. Такой идентификатор обычно называется **параметризованным типом**.

Листинг 2.69: Способы создания обобщённых идентификаторов и объектов

```
GBox<Integer> integerBox0;
GBox<Integer> integerBox1 = new GBox<Integer>(1);
GBox<Integer> integerBox2 = new GBox<>(1);
```

Чтобы создать экземпляр класса, используется ключевое слово `new` и, в дополнение, указывается, что создаётся не просто `GBox`, а обобщённый, поэтому пишется `<Integer>`. Компиляторы, начиная с Java 1.7, научились самостоятельно подставлять в треугольные скобки нужный тип (**выведение типа из контекста**).

Если тип совпадает с аргументом типа в идентификаторе, в скобках экземпляра его можно не писать. Это называется **бриллиантовый оператор**.

2.3.2. Варианты обобщений

Множество параметризированных типов

Ограничений на количество параметризированных типов не накладывается. Часто можно встретить обобщения с двумя типами, например, в коллекциях, хранящих пары ключ-значение. Также, нет ограничений на использование типов внутри угловых скобок.

Листинг 2.70: Множество параметризированных типов

```
private static class KVBox<K, V> {
   private K key;
   private V value;

   public KVBox(K key, V value) {
      this.key = key;
      this.value = value;
   }

   public V getValue() {
```

```
        return value;
    }
    public K getKey() {
        return key;
    }
    public void showType() {
        System.out.printf("Type of key is %s, key = %s, " +
                    "type of value is %s, value = %s\\n",
            key.getClass().getName(), getKey(),
            value.getClass().getName(), getValue());
    }
}

public static void main(String[] args) {
    KVBox<Integer, String> kvb0 = new KVBox<>(1, "Hello");
    KVBox<String, GBox<String>> kvb1 = new KVBox<>("World", new GBox<>("Java"));
}
```

Raw type (сырой тип)

Сырой тип – это имя обобщённого класса или интерфейса без аргументов типа, то есть это, фактически, написание идентификатора и вызов конструктора обобщённого класса как обычного, без треугольных скобок. При использовании сырых типов, программируется поведение, которое существовало до введения обобщений в Java.

Геттеры сырых типов возвращают объекты. Это логично, потому что ни на одном из этапов не указан аргумент типа.

GBox – это сырой тип обобщённого типа GBox<T>. Однако необобщённый класс или интерфейс не являются сырыми типами.

Для совместимости со старым кодом допустимо присваивать параметризованный тип своему собственному сырому типу.

Листинг 2.71: Использование сырых типов

```
GBox<Integer> intBox = new GBox<>(1);
GBox box = intBox;

GBox box = new GBox(1);
GBox<Integer> intBox = box;

GBox<Integer> intBox0 = new GBox<>(1);
GBox box0 = intBox0;
box.setValue(4);
```

Также, если присвоить параметризованному типу сырой тип, или если попытаться вызвать обобщённый метод в сыром типе, то буквально каждое слово в программе будет с предупреждением среды разработки. Предупреждения показывают, что сырой тип обходит проверку обобщённого типа, что откладывает обнаружение потенциальной ошибки на время выполнения программы.

Предупреждения среды, а на самом деле, предупреждения компилятора, обычно имеют вид, представленный на рис 2.20.

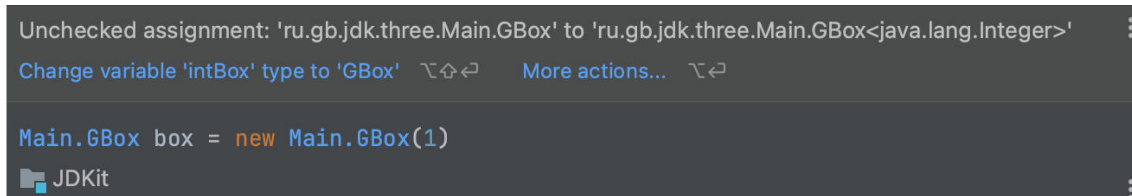


Рис. 2.20: Предупреждение среды о непроверенном типе

Термин «unchecked» означает непроверенные, то есть компилятор не имеет достаточного количества информации для обеспечения безопасности типов. По умолчанию этот вид предупреждений выключен, поэтому компилятор в терминале на самом деле даёт подсказку.

```
Note: <ClassName>.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Чтобы увидеть все «unchecked» предупреждения нужно перекомпилировать код с опцией `-Xlint:unchecked`.

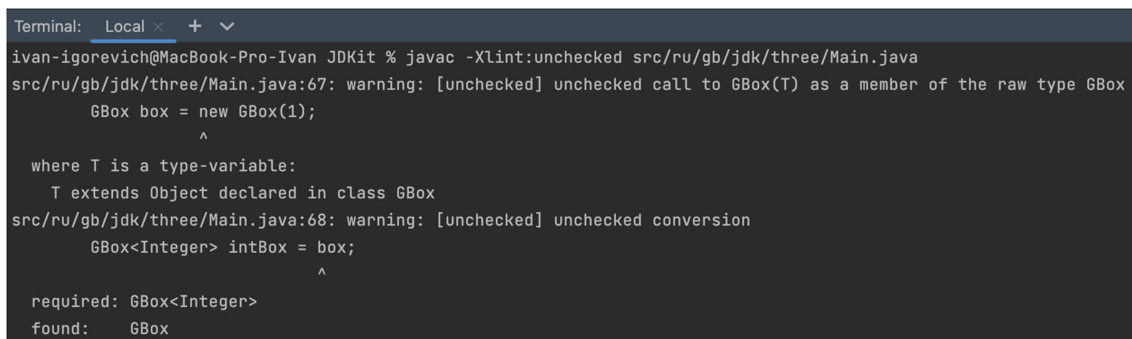


Рис. 2.21: Предупреждение компилятора о непроверенном типе

К предупреждениям среды в написанном коде желательно относиться внимательно, и придирчиво перепроверять код на наличие ненадёжных конструкций.

Вопросы для самопроверки

1. Обобщения – это способ создания общих
   (a) классов для разных пакетов;
   (b) алгоритмов для разных типов данных;
   (c) библиотек для разных приложений.
2. Что именно значит буква в угловых скобках?
   (a) Название обобщённого класса;
   (b) имя класса, с которым будет работать обобщение;
   (c) название параметра, используемого в обобщении.
3. Возможно ли передать обобщённым аргументом примитивный тип?
   (a) Да;
   (b) нет;
   (c) только строку.

Обобщённые методы

Обобщённые методы синтаксически схожи с обобщёнными классами, но параметры типа относятся к методу, а не к классу. Допустимо делать обобщёнными статические и не статические методы, а также конструкторы.

Синтаксис обобщённого метода включает параметры типа внутри угловых скобок, которые указываются перед возвращаемым типом.

Листинг 2.72: Обобщённый метод

```
private static <T> void setIfNull(GBox<T> box, T t) {
   if (box.getValue() == null) {
      box.setValue(t);
   }
}

public static void main(String[] args) {
   GBox<Integer> box = new GBox<>(null);
   setIfNull(box, 13);
   System.out.println(box.getValue());
   GBox<Integer> box0 = new GBox<>(1);
   setIfNull(box0, 13);
   System.out.println(box0.getValue());
```

2.3.3. Ограниченные параметры типа

Bounded type parameters позволяют ограничить типы данных, которые могут быть использованы в качестве параметров.

Использование bounded type parameters в Java является хорошей практикой, которая позволяет более точно определить используемые типы данных и обеспечивает более безопасный и читаемый код.

Ограничение «сверху»

В качестве примера необходимости ограничений будет использоваться уже написанная коробка. В коробке описывается операция сложения. Если сложение чисел и конкатенация строк – это понятные операции, то сложение котиков или потоков ввода-вывода не определены. Применение **bounded type parameters** позволяет более точно задавать ограничения на используемые типы данных, что помогает писать более безопасный и читаемый код. То есть, в обобщениях существует возможность ограничиться только теми типами, которые соответствуют определенным требованиям

Например, коробку предлагается сделать так, чтобы в ней хранились только числа – наследники класса `Number`. Подобное ограничение делается с помощью ограниченного параметра типа (bounded type parameter). Чтобы объявить «ограниченный сверху» параметр типа необходимо после имени параметра указать ключевое слово `extends`, а затем указать верхнюю границу (upper bound), которой в данном примере является класс `Number`.

Листинг 2.73: Ограничение параметра типа «сверху»

```
public class Box<V extends Number> {
```

```
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
    private V value;
}

private static <T extends Number> void setIfNull(BBox<T> box, T t) {
    if (box.getValue() == null) {
        box.setValue(t);
    }
}

public static void main(String[] args) {
    BBox<Integer> integerBBox = new BBox<>();
    BBox<String> stringBBox = new BBox<>();

    setIfNull(integerBBox, 4);
    setIfNull(stringBBox, "hello");
}
```

В рассматриваемом примере типы, которые можно использовать в параметризованных классах Box, ограничены наследниками класса Number. Если попытаться создать переменную с типом, например, Box<String>, то возникнет ошибка компиляции. Аналогичным образом создаются обобщённые методы с ограничением.

Листинг 2.74: Множественные ограничения

```
class Bird{}
interface Animal{}
interface Man{}
class CBox<T extends Bird & Animal & Man> {
    // ...
}
```

Также возможно задать несколько границ. При этом, важно помнить, что также, как и в объявлении классов, возможен только один класс-наследник и он указывается первым. Все остальные границы могут быть только интерфейсами и указываются через знак амерсанда.

В языке Java, несмотря на строгость типизации, возможно присвоить идентификатору одного типа объект другого типа, если эти типы **совместимы**. Например, можно присвоить объект типа Integer переменной типа Object, так как Object является одним из супертипов Integer. В объектно-ориентированной терминологии это называется связью «является» («is a»).

Предположим, что существует метод, описанный с generic параметром.

Листинг 2.75: Generic параметр метода

```
private static void boxTest(GBox<Number> n) { /* ... */ }
```

```
2
3  public static void main(String[] args) {
4     boxTest(new GBox<Number>(10));
5     boxTest(new GBox<Integer>(1)); // compile error
6     boxTest(new GBox<Float>(1.0f)); // compile error
```

На первый взгляд кажется, что в метод возможно передать `Box<Integer>` или `Box<Double>`, но нельзя, так как `Box<Integer>` и `Box<Double>` не являются потомками Box<Number>.

Это частое недопонимание принципов работы обобщений, и это важно знать. Наследование не работает в Java generics так, как оно работает в обычной Java.

Идентификатор коробки с `Number` не может в себе хранить коробку с `Integer`. Обобщение защищает от попыток положить в коробку, например, строк – не строку. То есть, предположим, в коробку кладётся `Integer`, как наследник `Number`, а затем, например `Float`, получится путаница.

Из приведённого примера возможно сделать вывод о том, что если методу с таким параметром передать коробку с `Integer` – он не будет работать. Чтобы допустить передачу таких контейнеров, в аргументе следует указать что в параметре возможен любой тип, являющийся наследником `Number`, то есть использовать маску `<? extends Number>`. Ограничивать такую маску возможно как сверху так и снизу. Таким образом, обобщения защищают самих себя от путаницы и не дают складывать в одни и те же контейнеры разные типы данных.

Поскольку коробку с чем то ещё, кроме `Number` и его наследников создавать нельзя, маски́рование при вызове метода будет избыточно

На самом деле обобщения – это так называемый синтаксический сахар. То есть, когда в коде используется обобщение, во время компиляции произойдёт так называемое «стирание», и все обобщённые типы данных преобразуются в `Object`, соответствующие проверки и приведения типов.

Ограничение «снизу»

Ограничение типов возможно вводить как сверху, так и снизу. На примере обобщённых методов. Такие методы необходимы, когда требуется объединить несколько похожих, но всё же разных типов данных. Если подать на вход обобщённого метода два типа – `Integer` и `Float` в итоге для работы будет выбран ближайший старший для них обоих – `Number`.

Листинг 2.76: Пример обобщённого метода

```
1  private static <T extends Number> boolean compare(T src, T dst) {
2     return src.equals(dst);
3  }
4
5  public static void main(String[] args) {
6     System.out.println(compare(1, 1.0f));
7     System.out.println(compare(1.0f, 1.0f));
8     System.out.println(compare(1, 1));
```

Например, даны два списка – один с `Integer` другой с `Number`, и требуется написать метод, который будет перекидывать числа из одного списка в другой. Для совершения этого

действия описан метод, при работе которого возможны два сценария – копировать элементы Number в список Integer или элементы Integer в список Number.

Листинг 2.77: Копирование чисел в списки

```
public static void copyTo(ArrayList src, ArrayList dst) {
   for (Object o : src) dst.add(o);
}

public static void main(String[] args) {
   ArrayList<Integer> ial = new ArrayList<>(Arrays.asList(1, 2, 3));
   ArrayList<Number> nal = new ArrayList<>(Arrays.asList(1f, 2, 3.0));
   System.out.println(ial);
   System.out.println(nal);
   copyTo(ial, nal);
   System.out.println(nal);
   copyTo(nal, ial);
   System.out.println(ial);
```

Правильным рабочим сценарием будет только второй – копирование более точного типа в более общий список. Java при компиляции пропустит в работу оба сценария, но действительно работающим всё равно остаётся только один.

Для примера, описаны два класса: «животное» и наследник животного – «кот». На их основе создаются обобщённые списки и вызывается обобщённый метод копирования списков.

Листинг 2.78: Копирование списков с «животными» и «котами»

```
public static void copyTo(ArrayList src, ArrayList dst) {
   for (Object o : src) dst.add(o);
}

private static class Animal {
   protected String name;
   protected Animal() { this.name = "Animal"; }
   @Override public String toString() { return name; }
}
private static class Cat extends Animal {
   protected Cat() { this.name = "Cat"; }
}

public static void main(String[] args) {
   ArrayList<Cat> cats = new ArrayList<>(Arrays.asList(new Cat()));
   ArrayList<Animal> animals = new ArrayList<>(Arrays.asList(new Animal()));
   copyTo(animals, cats);
   System.out.println(cats);
```

При компиляции и исполнении ошибок не возникло. К классу кота добавляется метод голос и совершается попытка вызвать голос у объекта из списка.

```
109            System.out.println(cats);
110            cats.get(1).voice();
111
112
Run:  ru.gb.jdk.three.Main
/Library/Java/JavaVirtualMachines/liberica-jdk-11.jdk/Contents/Home/bin/java -ja
[Cat, Animal]
Exception in thread "main" java.lang.ClassCastException Create breakpoint : class ru
    at ru.gb.jdk.three.Main.main(Main.java:110)

Process finished with exit code 1
```

Рис. 2.22: Проблема объединения списков

При попытке вызвать у более общего «животного» метод более частного «кота» выбрасывается исключение о невозможности приведения типов.

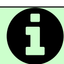
Для того, чтобы избежать подобных проблем, необходимо описать метод таким образом, чтобы он работал только с каким-то одним типом <T> и списки будут <T> и в цикле тоже будут перебираться элементы типа T. Если более не уточнять, получится, что в список котов возможно класть только котов, а в список животных класть только животных. Но кот – это наследник животного, его присутствие в списке животных уместно. Получается, что источником может быть список из заданного типа или его наследников, а приёмником – тип или его родители, и далее метод, виртуальная машина и другие механизмы сами разбираются, кто подходит под эти параметры.

При таком описании метода, неверные варианты будут отсекаться на этапе компиляции.

```
public static <T> void copyTo (
      ArrayList<? extends T> src, ArrayList<? super T> dst) {
   for (T o : src) {
      dst.add(o);
   }
}
```

2.3.4. Выведение типов

Алгоритм выведения типов определяет типы аргументов, а также, если это применимо, тип, в который присваивается результат или в котором возвращается результат.

 Выведение типов – это возможность компилятора автоматически определять аргументы типа на основе контекста.

Алгоритм работает от наиболее общего типа (`Object`) к наиболее точному, подходящему к данной ситуации. Например, добавив к коту реализацию интерфейса `Serializable` и написав метод, работающий с одним и только одним типом данных T будет создана ситуация, в которой никакие аргументы не связаны наследованием.

Листинг 2.79: Ситуация, в которой сработает выведение типов

```
   private static class Cat extends Animal implements Serializable {
      protected Cat() { this.name = "Cat"; }
      public void voice(){ System.out.println("meow"); }
   }
```

```
5     private static <T> T pick(T first, T second) { return second; }
6 
7     public static void main(String[] args) {
8         Serializable se1 = pick("d", new Cat());
9         Serializable se2 = pick("d", new ArrayList<String>());
```

В таком методе выведение типов определяет, что вторые аргументы метода `pick`, а именно `Cat` и `ArrayList`, передаваемые в метод имеют тип `Serializable`, но этого недостаточно, потому что первый аргумент тоже должен быть того же типа. Удачно, что строка – это тоже `Serializable`.

В описании обобщённых методов, выведение типа делает возможным вызов обобщённого метода так, будто это обычный метод, без указания типа в угловых скобках.

Листинг 2.80: Выведение типов в обобщённых методах

```
1  public class App {
2      public static <U> void addBox(U u, List<Box<U>> boxes) {
3          Box<U> box = new Box<>();
4          box.setValue(u);
5          boxes.add(box);
6      }
7 
8      public static void main( String[] args ) {
9          ArrayList<Box<Cat>> catsInBoxes = new ArrayList<>();
10         App.<Cat>addBox(new Cat("Kusya"), catsInBoxes);
11         addBox(new Cat("Kusya"), catsInBoxes);
12         addBox(new Cat("Murka"), catsInBoxes);
13         printBoxes(catsInBoxes);
14     }
```

Очевидно, что в листинге 2.80 обобщённый метод `addBox()` объявляет один параметр типа U. В большинстве случаев компилятор Java может вывести параметры типа вызова обобщённого метода, в результате чаще всего вовсе не обязательно их указывать.

Чтобы вызвать обобщённый метод `addBox()`, возможно указать параметры типа (строка 10) либо опустить их, тогда компилятор языка автоматически выведет тип Cat из аргументов метода при вызове (строка 11).

Выведение типа при создании экземпляра обобщённого класса позволяет заменить аргументы типа, необходимые для вызова конструктора обобщённого класса пустым множеством параметров типа (пустые треугольные скобки, бриллиантовая операция), так как компилятор может вывести аргументы типа из контекста.

Очевидно, что конструкторы могут быть обобщёнными как в обобщённых, так и в необобщённых классах.

Листинг 2.81: Обобщённый конструктор

```
1  public class Box<T> {
2      <U> Box(U u){
3          // ...
4      }
5      public T getValue() {
6          return value;
7      }
```

```
8      public void setValue(T value) {
9          this.value = value;
10     }
11     private T value;
12 }
13
14 public static void main(String[] args) {
15     Box<Cat> box = new Box<Cat>("Some message");
```

Например, возможно явно указать, что у коробки будет обобщённый аргумент «кот», а у конструктора – какой-то другой аргумент, например, строка или число. Компилятор выведет тип String для формального параметра U, так как фактически переданный аргумент является экземпляром класса String.

Алгоритм выведения типа использует только аргументы вызова, целевые типы и возможно очевидный ожидаемый возвращаемый тип для выведения типов. Алгоритм выведения не использует последующий код программы.

2.3.5. Целевые типы

Компилятор Java пользуется целевыми типами для вывода параметров типа вызова обобщённого метода.

Целевой тип выражения – это тип данных, который компилятор Java ожидает в зависимости от того, в каком месте находится выражение.

Например, как в методе `emptyBox()` (листинг 2.82, строка 12). В основном методе инициализация ожидает экземпляр `Box<String>`. Этот тип данных является целевым типом. Поскольку метод `emptyBox()` возвращает значение обобщённого типа `Box<T>`.

Листинг 2.82: Целевые типы

```
1  public class TBox<T> {
2      public static final TBox EMPTY_BOX = new TBox<>();
3
4      public T getValue() { return value; }
5
6      public void setValue(T value) {
7          this.value = value;
8      }
9
10     private T value;
11
12     static <T> TBox<T> emptyBox(){
13         return (TBox<T>) EMPTY_BOX;
14     }
15 }
16
17 public static void main( String[] args ) {
```

```
18    TBox<String> box = TBox.emptyBox();
19 }
```

2.3.6. Вопросы для самопроверки

1. Что из следующего является недопустимым?
   (a) `ArrayList<? extends Number> al1 = new ArrayList<Number>();`
   (b) `ArrayList<? extends Number> al2 = new ArrayList<Integer>();`
   (c) `ArrayList<? extends Number> al3 = new ArrayList<String>();`
   (d) Всё допустимо.
2. параметры метода `ArrayList<? extends T> src, ArrayList<? super T> dst` вызов метода `copyTo(cats, animals);` Какой тип данных будет взят в качестве T?
   (a) Animal;
   (b) Cat;
   (c) Object.

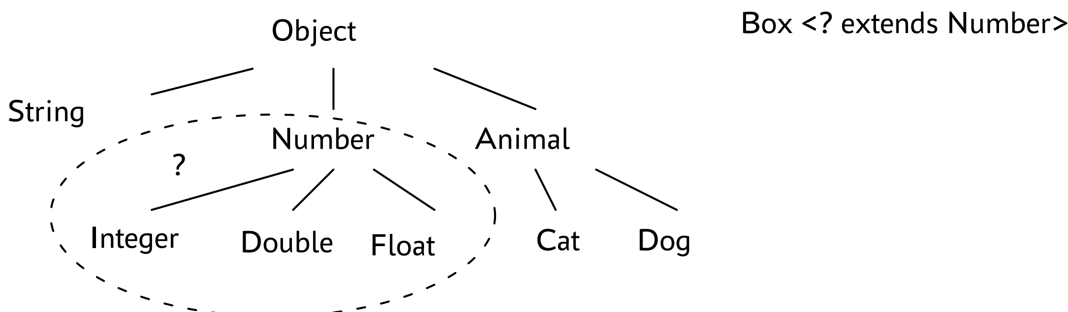
2.3.7. Подстановочный символ <?> (wildcard)

В обобщённом коде знак вопроса, называемый подстановочным символом, означает неизвестный тип. Подстановочный символ может использоваться в разных ситуациях: как параметр типа, поля, локальной переменной, иногда в качестве возвращаемого типа. Подстановочный символ никогда не используется (рис 2.23) в качестве аргумента типа для вызова обобщённого метода, создания экземпляра обобщённого класса или супертипа.

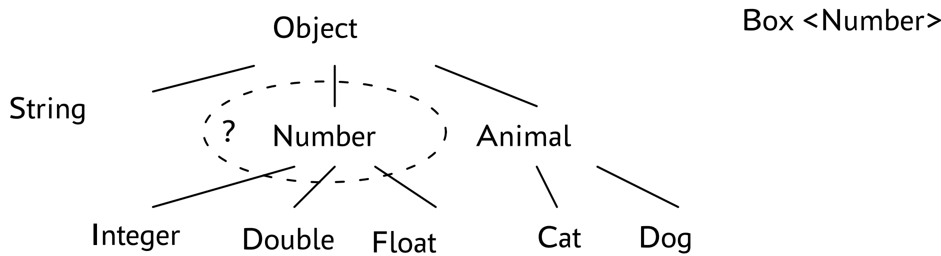
```
119
120     private static <?> T method(T first, T second) { return second; }
121     public static void main(String[] args) {
122         TBox<String> b = new TBox<?>();
123
```

Рис. 2.23: Ошибки при работе с подстановочным символом

Передаваемые типы для уточнения, возможно **ограничивать** сверху и снизу. Ограниченный сверху подстановочный символ применяется, чтобы **ослабить ограничения** переменной.



Чтобы написать метод, который работает с коробками, в которых содержится `Number` и дочерние от `Number` типы, например `Integer`, `Double` и другие, необходимо указать `Box<? extends Number>`.



Пример кода будет приводиться с классами животного и наследников.

Листинг 2.83: Подготовка классов для демнстрации

```
private static class Animal {
   protected String name;
   protected Animal(String name) { this.name = name; }
   @Override public String toString() {
      return this.getClass().getSimpleName() + " with name " + name;
   }
}
private static class Cat extends Animal {
   protected Cat(String name) { super(name); }
}
private static class Dog extends Animal {
   protected Dog(String name) { super(name); }
}
```

Внутри метода, выводящего информацию о содержимом коробки присутствует ограниченный сверху подстановочный символ `<? extends Animal>`, где вместо `Animal` может быть любой тип.

Листинг 2.84: Использование ограниченного подстановочного символа

```
public static class TBox<T> {
   public static final TBox EMPTY_BOX = new TBox<>();
   private T value;

   public T getValue() { return value; }
   public void setValue(T value) { this.value = value; }
   static <T> TBox<T> emptyBox() {
      return (TBox<T>) EMPTY_BOX;
   }
   @Override public String toString() {
      return value.toString();
   }
}
static void printInfo(TBox<? extends Animal> animalInBox) {
   System.out.println("Information about animal: " + animalInBox);
}
```

Создав соответствующие объекты, положив их в коробки и запустив код возможно наблюдать, что коробка вмещает как животных, так и наследников животного, чего не произошло бы при использовании в параметре типа животного без подстановочного символа.

Если просто использовать подстановочный символ <?>, то получится подстановочный символ без ограничений. Box<?> означает коробку с неизвестным содержимым (неизвестным типом).

Такой синтаксис существует для того, чтобы продолжать использовать обобщённый тип без уточнения типа, как следствие, без использования обобщённой функциональности. Неограниченный подстановочный символ полезен, если нужен метод, который может быть реализован с помощью функциональности класса `Object`. Когда код использует методы обобщённого класса, которые не зависят от параметра типа.

В программах, использующих Reflection API конструкция `Class<?>` используется чаще других конструкций, потому что большинство методов объекта Класс не зависят от расположенного внутри типа.

Например, метод `printInfo()` (строка 14) из листинга 2.84, не использует никаких методов животного, цель метода – вывод в консоль информации об объекте в коробке любого типа, поэтому в параметре метода можно заменить «коробку с наследниками животного» на «коробку с чем угодно», ведь в методе будет использоваться только метод коробки `toString()`.

Листинг 2.85: Исправление метода вывода информации на экран

```
1 static void printInfo(TBox<?> animalInBox) {
2     System.out.println("Information about animal: " + animalInBox);
3 }
```

Важно запомнить, что `Box<Object>` и `Box<?>` – это не одно и то же.

```
127
128     static void printInfo(TBox<Object> animalInBox){
129         System.out.println("Information about animal: " + animalInBox);
130     }
131
132     public static void main(String[] args) {
133         TBox<Cat> catInBox = TBox.emptyBox();
134         catInBox.setValue(new Cat("Vasya"));
135         printInfo(catInBox);
136                        Required type:  TBox <Object>
137         TBox<Dog> dogIn   Provided:   TBox <Cat>
138         dogInBox.setVal   Change 1st parameter of method 'printInfo' from 'TBox<Object>' to 'TBox<Cat>'   More actions...
139         printInfo(dogIn   Main.TBox<Main.Cat> catInBox = TBox.emptyBox()
                              JDKit
```

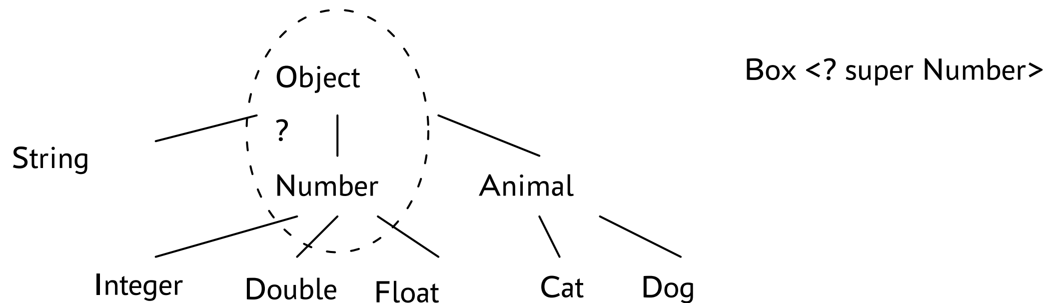
Рис. 2.24: Ошибка компиляции при использовании параметра типа `Object`

Ограниченный снизу подстановочный символ ограничивает неизвестный тип так, чтобы он был либо указанным типом, либо одним из его предков.

В обобщённых конструкциях возможно указать либо только верхнюю границу для подстановочного символа, либо только нижнюю.

```
127
128    static void printInfo(TBox<? extends Number & ? super Integer> animalInBox){
129        System.out.println("Information about a   '>' or ',' expected
130    }                                             Identifier expected
131                                                  java.lang
```

То есть, если написать метод `printInfo()`, с параметром коробки и обобщённым аргументом не `<? extends Animal>`, а `<? super Animal>`, токод также не будет работать, потому что метод будет ожидать не «животное и наследников», а «животное и родителей», то есть `Object`.

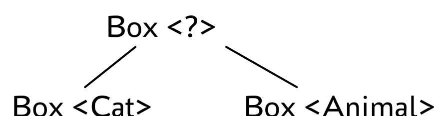


Обобщённые классы или интерфейсы связаны не только из-за связи между их типами. С обычными, необобщёнными классами, наследование работает по правилу подчинённых типов: класс `Cat` является подклассом класса `Animal`, и расширяет его.

Листинг 2.86: Наследование необобщённых классов

```
private static class Animal {
    protected String name;
    protected Animal(String name) { this.name = name; }
    @Override public String toString() {
        return this.getClass().getSimpleName() + " with name " + name;
    }
}
private static class Cat extends Animal {
    protected Cat(String name) { super(name); }
}

public static void main(String[] args) {
    Cat cat = new Cat("Vasya");
    Animal animal = cat;

    TBox<Cat> catInBox = new TBox<>();
    TBox<Animal> animalInBox = catInBox; // Incompatible types
```

Данное правило не работает для обобщённых типов. Несмотря на то, что `Cat` является подтипом `Animal`, `Box<Cat>` не является подтипом `Box<Animal>`. Общим предком для `Box<Animal>` и `Box<Cat>` является `Box<?>`.



Листинг 2.87: Наследование в параметрах типа через подстановочнвй символ

```
TBox<? extends Cat> catInBox = new TBox<>();
TBox<? extends Animal> animalInBox = catInBox; // OK
```

В некоторых случаях компилятор может вывести тип подстановочного символа. Коробка может быть определена как `Box<?>`, но при вычислении выражения компилятор выведет конкретный тип из кода, такой сценарий называется **захватом подстановочного символа**. В большинстве случаев нет нужды беспокоиться о захвате подстановочного символа, кроме случаев, когда в сообщении об ошибке появляется фраза `capture of`. В примере, приведённом в листинге 2.88 компилятор обрабатывает параметр коробки как тип `Object`. Когда вызывается метод внутри `testError()`, компилятор не может подтвердить тип объекта, который будет присутствовать внутри коробки и генерирует ошибку. Обобщения были добавлены в Java именно для этого – чтобы усилить безопасность типов на этапе компиляции.

Листинг 2.88: Ошибка захвата подстановочного символа

```
static void testError(Box<?> box){
    box.setValue(box.getValue()); // capture of ?
}
```

Когда есть уверенность в том, что операция безопасна, ошибку возможно исправить написав приватный вспомогательный метод (в англоязычной литературе private helper method), захватывающий подстановочный символ.

Листинг 2.89: Использование вспомогательного метода

```
private static <T> void testErrorHelper(TBox<T> box) {
    box.setValue(box.getValue());
}

static void testError(TBox<?> box) {
    testErrorHelper(box);
}
```

Краткое руководство по использованию подстановочных символов

Входная переменная. Предоставляет данные для кода. Для метода `copy(src, dst)` параметр `src` предоставляет данные для копирования, поэтому он считается входной переменной.

Выходная переменная. Содержит данные для использования в другом месте. В примере `copy(src, dst)` параметр `dst` принимает данные и будет считаться выходной переменной.

1. Входная переменная определяется с ограничением сверху.
2. Выходная переменная определяется с ограничением снизу.
3. Если ко входной переменной можно обращаться только как к `Object` – неограниченный подстановочный символ.
4. Если переменная должна использоваться как входная и как выходная одновременно, НЕ использовать подстановочный символ.
5. Не использовать подстановочные символы в возвращаемых типах.

Вопросы для самопроверки

1. Ограниченный снизу подстановочный символ позволяет передать в качестве аргумента типа
   (a) только родителей типа;
   (b) только наследников типа;
   (c) сам тип и его родителей;
   (d) сам тип и его наследников.
2. Конструкция <? super Object>
   (a) не скомпилируется;
   (b) не имеет смысла;
   (c) не позволит ничего передать в аргумент типа;
   (d) не является чем-то необычным.

2.3.8. Стирание типа и загрязнение кучи

Обобщения были введены в язык программирования Java для обеспечения более жёсткого контроля типов во время компиляции и для поддержки обобщённого программирования. Для реализации обобщения компилятор Java применяет стирание типа.

Стирание типа

- Механизм стирания типа фактически заменяет все параметры типа в обобщённых типах их границами или `Object`, если параметры типа не ограничены.
- Сгенерированный байткод содержит только обычные классы, интерфейсы и методы.
- Вставляет явное приведение типов где необходимо.
- Генерирует связующие методы, чтобы сохранить полиморфизм в расширенных обобщённых типах.
- Гарантирует, что никакие новые классы не будут созданы для параметризованных типов, следовательно обобщения не приводят к накладным расходам во время исполнения.

Компилятор Java также стирает параметры типа обобщённых методов. Обобщённый метод в котором используется неограниченный тип будет заменён компилятором на `Object`.

Аналогично классам для методов происходит стирание типа при расширении ключевым словом `extends` – параметр в угловых скобках заменяется на максимально возможного родителя.

```
private static <T> void setIfNull(TBox<T> box, T t) {
    if (box.getValue() == null) {
        box.setValue(t);
    }
}
// ... both methods have same erasure
private static void setIfNull(TBox<Object> box, Object t) {
    if (box.getValue() == null) {
        box.setValue(t);
    }
}
```

Стирание типа имеет последствия, связанные с произвольным количеством параметров (varargs).

Материализуемые типы – это типы, информация о которых полностью доступна во время выполнения, такие как примитивы, необобщённые типы, сырые типы, обращения к неограниченным подстановочным символам.

Нематериализуемые типы – это типы, информация о которых удаляется во время компиляции стиранием типов, например, обращения к обобщённым типам, которые не объявлены с помощью неограниченных подстановочных символов. Во время выполнения о нематериализуемых типах нет всей информации. Виртуальная машина Java не может узнать разницу между нематериализуемыми типами во время выполнения.

Загрязнение кучи

Загрязнение кучи (heap pollution) возникает, когда переменная параметризованного типа ссылается на объект, который не является параметризованным типом. Такая ситуация возникает, если программа выполнила операцию, генерирующую предупреждение `unchecked warning` во время компиляции. Предупреждение `unchecked warning` генерируется, если правильность операции, в которую вовлечён параметризованный тип (например, приведение типа или вызов метода) не может быть проверена.

Если компилируются различные части кода отдельно, то становится трудно определить потенциальную угрозу загрязнения кучи.

2.3.9. Ограничения обобщений (резюме)

В этом разделе приводится некоторое резюме вышесказанного в части наиболее частых ошибок при работе с обобщениями.

Нельзя использовать при создании экземпляра примитивы. Выход из ситуации – использование классов-обёрток.

```
155
156        TBox<int> box = new TBox<>();
157
```

Нельзя создавать экземпляры параметров типа. В качестве выхода из ситуации – передавать вновь созданный объект в обобщённые методы в качестве параметра.

```
149
150 @    static <T> void add(Box<T> box) {
151          T t = new T();
152          // ...
153          box.setValue(t);
154      }
155
```

Статические поля класса являются общими для всех объектов этого класса, поэтому статические поля с типом параметра типа запрещены. Так как статическое поле является общим для коробки с животным, коробки с котом и коробки с собакой, то какого типа это поле? Также запрещено использовать приведение типа к параметризованному типу, если он не использует неограниченный подстановочный символ.

```
149
150     public class SBox<T> {
151         private static T value;
152
153         // ...
154     }
155
156
```

Static declarations in inner classes are not supported at language level '11'
Upgrade JDK to 16+ More actions...
ru.gb.jdk.three.Main.SBox<T>
private static T value
JDKit

Запрещено использовать приведения типов для обобщённых объектов. Так как компилятор стирает все параметры типа из обобщённого кода, то нельзя проверить во время выполнения, какой параметризованный тип используется для обобщённого типа.

```
163
164         TBox<Integer> box1 = new TBox<>();
165         TBox<Number>  box2 = (TBox<Number>) box1;
166
```

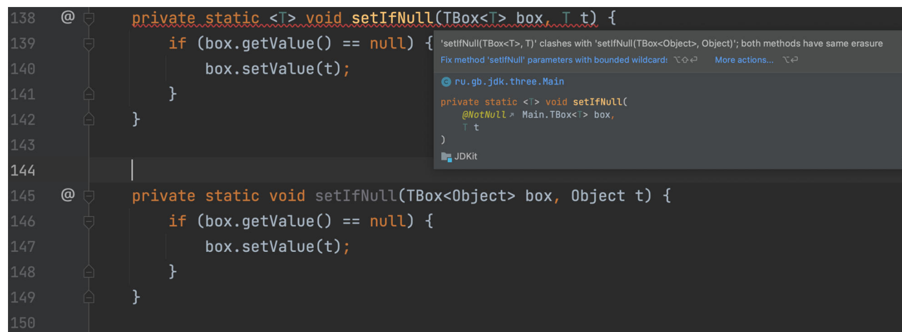
Запрещено создавать массивы параметризованных типов.

```
159
160         Object[] stringLists = new TBox<String>[];
161         // compilation error, but let's say it's possible
162         stringLists[0] = new TBox<String>();   // OK
163         stringLists[1] = new TBox<Integer>();  // here should be
164                                  // an ArrayStoreException exception,
165                                  // but the runtime cannot notice it.
166
```

Обобщённый класс не может расширять класс `Throwable` напрямую или опосредовано. Метод не может ловить (`catch`) экземпляр параметра типа. Однако можно использовать параметр типа в `throws`.

```
157     // Extends Throwable non-direct
158     class MathException<T> extends Exception { /* ... */ } // compilation error
159
160     // Extends Throwable directly
161     class QueueFullException<T> extends Throwable { /* ... */ // compilation error
162
163         public static <T extends Exception, J> void execute(TBox<J> box) {
164             try {
165                 J j = box.getValue();
166                 // ...
167             } catch (T e) {   // compilation error
168             }
169         }
170
171         class Parser<T extends Exception> {
172             public void parse(File file) throws T { // OK
173                 // ...
174             }
175         }
```

Класс не может иметь два перегруженных метода, которые будут иметь одинаковую сигнатуру после стирания типов. Такой код не скомпилируется.



Практическое задание

1. Написать метод, который меняет два элемента массива местами (массив может быть любого ссылочного типа);
2. Большая задача:
    - Есть классы `Fruit` -> `Apple`, `Orange`; (больше не надо);
    - Класс `Box` в который можно складывать фрукты, коробки условно сортируются по типу фрукта, поэтому в одну коробку нельзя сложить и яблоки, и апельсины; Для хранения фруктов внутри коробки можете использовать `ArrayList`;
    - Сделать метод `getWeight()` который высчитывает вес коробки, зная количество фруктов и вес одного фрукта(вес яблока – `1.0f`, апельсина – `1.5f`, не важно в каких единицах);
    - Внутри класса коробки сделать метод `compare()`, который позволяет сравнить текущую коробку с той, которую подадут в `compare()` в качестве параметра, `true` – если их веса равны, `false` в противном случае (коробки с яблоками возможно сравнивать с коробками с апельсинами);
    - Написать метод, который позволяет пересыпать фрукты из текущей коробки в другую коробку (при этом, нельзя яблоки высыпать в коробку с апельсинами), соответственно, в текущей коробке фруктов не остается, а в другую перекидываются объекты, которые были в этой коробке.

Термины, определения и сокращения

`finally`	часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.
`throws`	ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.
`throw`	оператор, активирующий (выбрасывающий) объект исключения.
`try...catch`	двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.
BIOS	(англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.
CI	(англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.
CLI	(англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».
cp1251	набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.
Docker	программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.
Полиморфизм	это возможность объектов с одинаковой спецификацией иметь различную реализацию
Потоки в-в	объекты, из которых можно считать данные и в которые можно записывать данные
Разделы	(англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.
Сборщик мусора	специализированная подпрограмма, очищающая память от неиспользуемых объектов.
События	это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.
Статика	(статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.
Стек	структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.
Строка	ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.
Типизация	классификация по типам
ФС	Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

Семинары

A. Семинар: компиляция и интерпретация кода

A.1. Инструментарий

– Презентация для преподавателя, ведущего семинар;
– Фон GeekBrains для проведения семинара в Zoom;
– Jupyter Notebook для практики и примеров используется Jupyter notebook (потребуется установить Python и ядро IJava) и любой терминал операционной системы (bash, zsh, cmd);
– JDK любая 11 версии и выше;
– Docker, make.

A.2. Цели семинара

– Закрепить полученные на лекции знания, касающиеся компиляции, интерпретации кода и создания программной документации;
– Получить практический навык настройки терминала ОС для компиляции и исполнения кода, установки сторонних библиотек для интерпретации;
– Попрактиковаться в написании терминальных команд и простых проектов. Лучше понять принцип импортирования кода и сборки проекта.

A.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-4	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучиает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	3-14	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (4 вопроса, по минуте на ответ)
Рассмотрение ДЗ	15	15-18	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов

Что происходит	Время	Слайды	Описание
Вопросы и ответы	10	19	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	10	20-22	Создать, скомпилировать, запустить и продемонстрировать простой проект без использования среды разработки. Показать выполненные терминальные команды, результат компиляции. (* отделить исходный код от скомпилированных файлов, ** сложить исходный код в пакет)
Перерыв (если нужен)	5	26	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 2	10	23-25	Настроить окружение Jupyter Notebook с ядром Java, создать одну ячейку с переменной, а вторую с выводом значения этой переменной стандартным способом. Дополнить ячейки описанием markdown. (* осуществить форматированный вывод, ** сохранить форматирующую строку в ячейке с переменной)
Задание 3	15	27-29	К проекту из задания 1 добавить ещё один класс в соседнем пакете, как это было показано на лекции и комментарии в стиле Javadoc. Комментарии необходимо добавить как к методам, так и к классам. Сгенерировать программную документацию. (* создать документацию на каждый пакет по отдельности)
Домашнее задание	5	39	Объясните домашнее задание, подведите итоги урока
Рефлексия	10	40-42	Преподаватель запрашивает обратную связь
Длительность	90		

A.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Презентует название курса (первый раз) и семинара;
  - Рассказывает немного о себе;
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5-7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**

   - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
   - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Какой механизм используется для непосредственного исполнения скомпилированного кода? (3)
     (a) JDK;
     (b) JRE;
     (c) JVM.
  2. Какая сущность только объединяет классы по смыслу? (1)
     (a) Пакеты;
     (b) Библиотеки;
     (c) Фреймворки.
  3. Основная единица исходного кода программы – это? (2)
     (a) Функция;
     (b) Класс;
     (c) Файл.
  4. Какой ключ используется для указания папки назначения? (1)
     (a) `-d`;
     (b) `-out`;
     (c) `-to`.

Рассмотрение ДЗ

- **Цель этапа:** Пояснить неочевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
   - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;
   - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
   - Создать проект из трёх классов (основной с точкой входа и два класса в другом пакете), которые вместе должны составлять одну программу, позволяющую производить четыре основных математических действия и осуществлять форматированный вывод результатов пользователю.
     Вариант решения

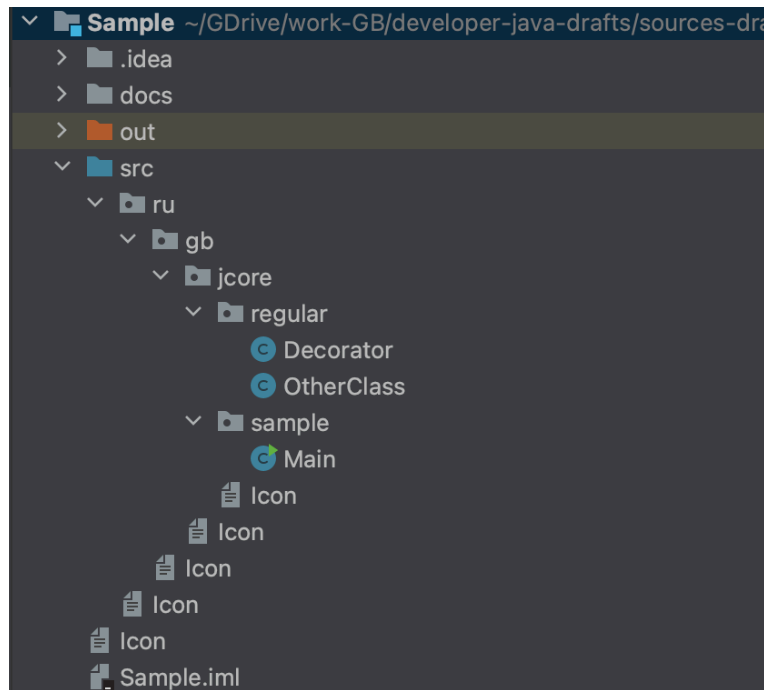


Рис. 25: Структура проекта

Листинг 90: Код основного класса

```
package ru.gb.jcore.sample;

import ru.gb.jcore.regular.Decorator;
import ru.gb.jcore.regular.OtherClass;

/**
* Основной класс приложения. Здесь мы можем описать
* его основное назначение и способы взаимодействия с ним.
* */
public class Main {
  /**
  * Точка входа в программу. С неё всегда всё начинается.
  *
  * @param args стандартные аргументы командной строки
  * */
  public static void main(String[] args) {
    int result = OtherClass.add(2, 2);
    System.out.println(Decorator.decorate(result));
    result = OtherClass.sub(2, 2);
    System.out.println(Decorator.decorate(result));
    result = OtherClass.mul(2, 2);
    System.out.println(Decorator.decorate(result));
    result = OtherClass.div(2, 2);
    System.out.println(Decorator.decorate(result));
  }
}
```

Листинг 91: Код считающего класса

```
package ru.gb.jcore.regular;

/**
 * Другой, очень полезный класс приложения. Здесь мы можем описать
 * его основное назначение и способы взаимодействия с ним.
 * */
public class OtherClass {
   /**
    * Функция суммирования двух целых чисел
    *
    * @param a первое слагаемое
    * @param b второе слагаемое
    * @return сумма a и b, без проверки на переполнение переменной.
    * */
   public static int add(int a, int b) {
      return a + b; // возврат без проверки переполнения
   }

   /**
    * Функция деления двух целых чисел
    *
    * @param a делимое
    * @param b делитель
    * @return частное a и b, без проверки на переполнение переменной.
    * */
   public static int div(int a, int b) {
      return a / b; // возврат без проверки переполнения
   }

   /**
    * Функция умножения двух целых чисел
    *
    * @param a первый множитель
    * @param b второй множитель
    * @return произведение a и b, без проверки на переполнение
      переменной.
    * */
   public static int mul(int a, int b) {
      return a * b; // возврат без проверки переполнения
   }

   /**
    * Функция вычитания двух целых чисел
    *
    * @param a уменьшаемое
    * @param b вычитаемое
    * @return разность a и b, без проверки на переполнение переменной.
    * */
   public static int sub(int a, int b) {
      return a - b; // возврат без проверки переполнения
```

```
    }
}
```

Листинг 92: Код декоратора

```
package ru.gb.jcore.regular;

/**
 * Декоратор. Он декорирует, то есть, накладывает на результат
   декорации.
 * Внешний вид важен, поэтому этот класс существует.
 * */
public class Decorator {
    /**
     * Функция декорирования числа для вывода в консоль
     * в виде строки с преамбулой 'Вот Ваше число'
     *
     * @param a число, требующее декорации
     * @return Отформатированная строка.
     * */
    public static String decorate(int a) {
        /*
         * Метод декорирует число, добавляя к нему строку
         * при помощи функции форматирования строк
         * */
        return String.format("Here is your number: %d.", a);
    }
}
```

- Скомпилировать проект, а также создать для этого проекта стандартную веб-страницу с документацией ко всем пакетам.
  Вариант решения

Листинг 93: Команды компиляции

```
javac -sourcepath ./src -d out src/ru/gb/jcore/sample/Main.java
java -classpath ./out ru.gb.jcore.sample.Main

```

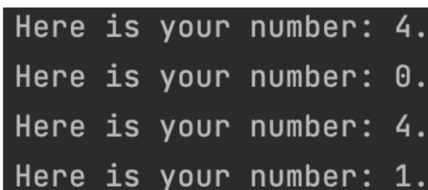


Рис. 26: Результат компиляции

Листинг 94: Команда создания документации

```
javadoc -d docs -sourcepath ./src -cp ./out -subpackages ru

```

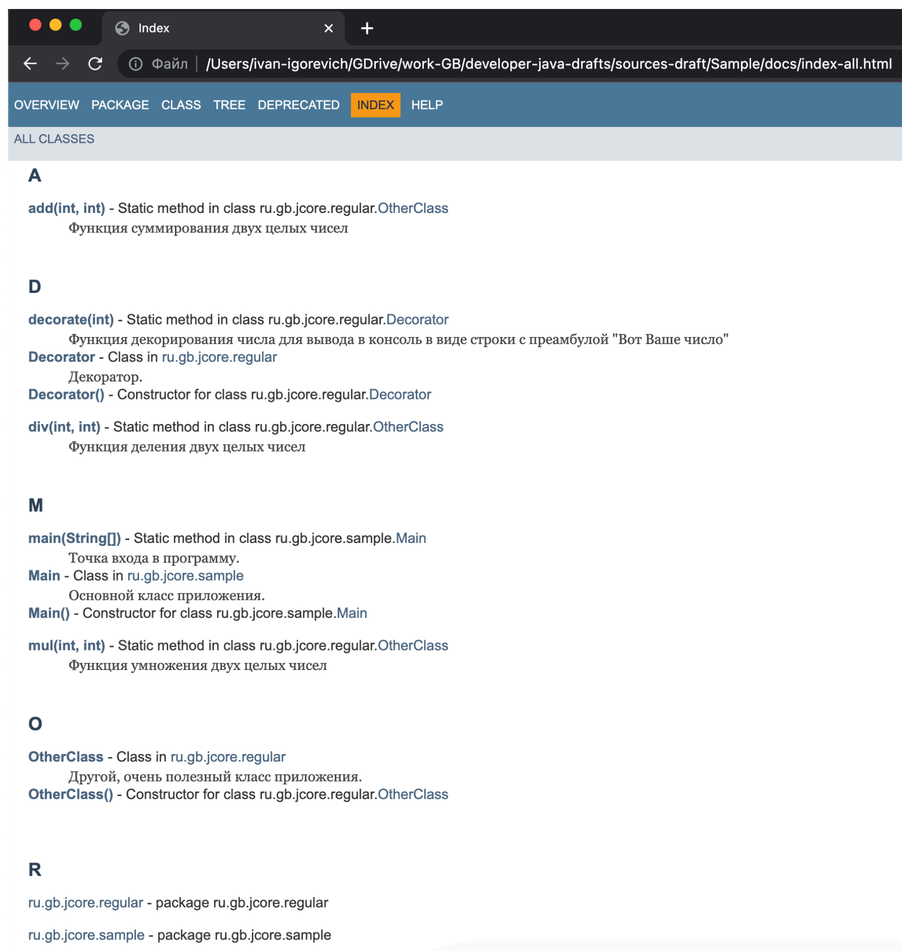


Рис. 27: Результат создания документации

- Создать Makefile с задачами сборки, очистки и создания документации на весь проект.
  Вариант решения[12]

Листинг 95: Makefile

```
SRC_DIR := src
OUT_DIR := out
DOC_DIR := doc

JC := javac
JDOC := javadoc
JSRC := -sourcepath ./$(SRC_DIR)
JCLASS := -cp ./$(OUT_DIR)
JCDEST := -d $(OUT_DIR)
JDOCDEST := -d $(DOC_DIR)
MAIN_SOURCE := ru/gb/jcore/sample/Main
MAIN_CLASS := ru.gb.jcore.sample.Main

all:
	${JC} ${JSRC} ${JCDEST} ${SRC_DIR}/${MAIN_SOURCE}.java
```

[12] Обратите внимание, что все отступы сделаны не пробелами, а табуляцией, иначе Makefile не работает

```
clean:
    rm -R ${OUT_DIR}

run:
    cd out && java ${MAIN_CLASS}

docs:
    ${JDOC}) ${JDOCDEST} ${JSRC} ${JCLASS} -subpackages ru
```

```
ivan-igorevich@MacBook-Pro-Ivan Sample % make all
javac -sourcepath ./src -d out src/ru/gb/jcore/sample/Main.java
ivan-igorevich@MacBook-Pro-Ivan Sample % make run
cd out && java ru.gb.jcore.sample.Main
Here is your number: 4.
Here is your number: 0.
Here is your number: 4.
Here is your number: 1.
ivan-igorevich@MacBook-Pro-Ivan Sample % make docs
make: `docs' is up to date.
ivan-igorevich@MacBook-Pro-Ivan Sample % make clean
rm -R out
ivan-igorevich@MacBook-Pro-Ivan Sample %
```

Рис. 28: Результат выполнения тасков

- *Создать два Docker-образа. Один должен компилировать Java-проект обратно в папку на компьютере пользователя, а второй забирать скомпилированные классы и исполнять их.
  Вариант решения
  Для упрощения был использован docker compose, вместо чистого Docker. Файлы, компилирующие и исполняющие программу представлены в листингах ниже. Оба эти файла запускаются из корня папки проекта командами

  ```
  docker compose -f docker-compose-class.yml up
  docker compose -f docker-compose-exec.yml up
  ```

  соответственно.

Листинг 96: docker-compose-class.yml

```
services:
  app:
    image: bellsoft/liberica-openjdk-alpine:11.0.16.1-1
    command: javac -sourcepath /app/src -d /app/out /app/src/ru/gb/
   jcore/sample/Main.java
    volumes:
      - ./bin:/app/out
      - ./src:/app/src
```

Листинг 97: docker-compose-exec.yml

```
services:
  app:
    image: bellsoft/liberica-openjdk-alpine:11.0.16.1-1
    command: java -classpath /app/out ru.gb.jcore.sample.Main
```

```
5     volumes:
6       - ./bin:/app/out
```

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
  - Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
  - Посоветовать литературу на тему заданного вопроса;
  - Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Создание, компиляция и запуск проектов без использования среды разработки.
- **Тайминг** 10-20 минут.
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (запуск приложений на сервере, в контейнерах, настройка CI/CD в пет-проектах);
  - Выдать задание группам студентов, показать где именно следует искать терминал ОС;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**:
  - Создать, скомпилировать, запустить и продемонстрировать простой проект без использования среды разработки.

    Вариант решения

Листинг 98: Простейший проект

```
1 public class Main {
2    public static void main(String[] args) {
3       System.out.println("Hello, world!");
4    }
5 }
```

Листинг 99: Команды компиляции

```
1 javac Main.java
2 java Main
3
```

$*_1$ отделить исходный код от скомпилированных файлов
Вариант решения

```
javac -d out Main.java
java -classpath ./out Main

```

$*_2$ сложить исходный код в пакет с глубиной иерархии не менее 3.
Вариант решения
Вручную создать соответствующие вложенные папки, переместить в них файл с исходным кодом `Main.java` и написать оператор `package` первой строкой файла `Main.java`.

```
javac -d out ru/gb/jcore/Main.java
java -classpath ./out ru.gb.jcore.Main

```

Задание 2

- **Ценность этапа** Настройка и изучение дополнительного инструментария для создания проектов и описания работы фрагментов кода в виде Jupyter notebook.
- **Тайминг** 10-15 минут.
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (использование скриптовых возможностей среды разработки, таких как написание простых фрагментов кода без необходимости создавать большой проект в тяжеловесной среде разработки);
  - Пояснить студентам в каком виде выполнять и сдавать задания;
  - Выдать задание группам студентов, показать где и как скачивать необходимый инструментарий, если он ещё не установлен;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**
  - Настроить окружение Jupyter Notebook с ядром IJava, создать одну ячейку с переменной, а вторую с выводом значения этой переменной стандартным способом. Дополнить ячейки описанием markdown.

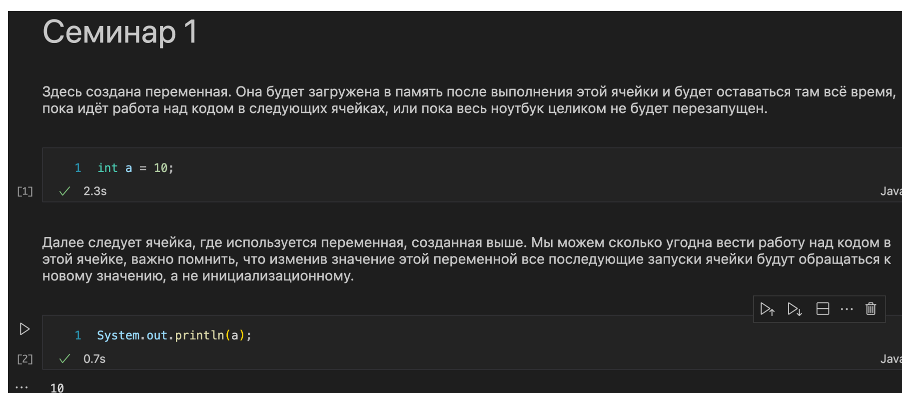


Рис. 29: Вариант решения

$*_1$ осуществить форматированный вывод

Листинг 100: Вариант решения (вторая ячейка)

```
System.out.print("Your number is " + a);
```

$*_2$ сохранить форматирующую строку в ячейке с переменной

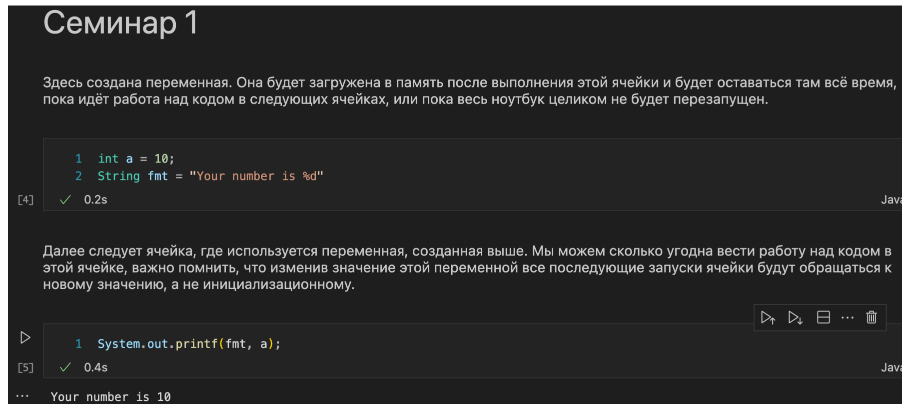


Рис. 30: Вариант решения

Задание 3

- **Ценность этапа** Закрепление навыков создания стандартной программной документации.
- **Тайминг** 15-20 минут
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (описание пет-проектов для потенциальных соисполнителей, создание базы знаний по проекту на случай длительных пауз в работе)
  - Выдать задание группам студентов
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдать группе задание «со звёздочкой».
- **Задания**
  - К проекту из задания 1 добавить ещё один класс в соседнем пакете, как это было показано на лекции, и комментарии в стиле Javadoc. Комментарии необходимо добавить как к методам, так и к классам. Сгенерировать общую программную документацию.

    Вариант решения

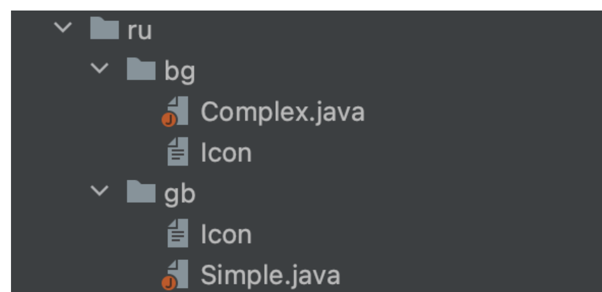


Рис. 31: Иерархия получившегося проекта

Листинг 101: Основной класс

```java
package ru.gb;

import ru.bg.Complex;
/**
 * Это простой класс. Он простой настолько, что ничего не делает
 */
public class Simple {
    /**
     * Функция запускающая программу и приветствующая мир.
     * Наверное, самая популярная функция в мире.
     *
     * @param args стандартные аргументы командной строки
     * */
    public static void main(String[] args) {
        System.out.println(Complex.hello());
    }
}
```

Листинг 102: Вспомогательный класс

```java
package ru.bg;

/**
 * Это уже весьма усложнённый класс, мы будем его вызывать из простого
 */
public class Complex {
    /**
     * Функция, возвращающая какую-то строку. Возможно, даже
       приветствующую мир.
     *
     * @return строка с приветствием.
     * */
    public static String hello() {
        return "Hello, world!";
    }
}
```

Листинг 103: Команды компиляции и создания документации

```
javac -sourcepath . -d out ru/gb/Simple.java
java -classpath ./out ru.gb.Simple
javadoc -d doc -sourcepath . -cp ./out -subpackages ru

```

$*_1$ создать документацию на каждый пакет по отдельности

Листинг 104: Вариант решения

```
javadoc -d doc_gb -sourcepath . -cp ./out ru.gb
javadoc -d doc_bg -sourcepath . -cp ./out ru.bg

```

Домашнее задание

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки
  - Объяснить где взять проект заготовки для дз
- **Задания**

5-25 мин Решить все задания (в том числе «со звёздочкой»), если они не были решены на семинаре, без ограничений по времени;
Все варианты решения приведены в тексте семинара выше

10-15 мин Создать docker-контейнер для формирования полной документации по проекту

Листинг 105: docker-compose-class.yml

```
services:
  app:
    image: bellsoft/liberica-openjdk-alpine:11.0.16.1-1
    command: javadoc -sourcepath /app/src -cp /app/out -d /app/doc -subpackages ru
    volumes:
      - ./bin:/app/out
      - ./src:/app/src
      - ./doc:/app/doc
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

Б. Семинар: данные и функции

Б.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

Б.2. Цели семинара

- Закрепить полученные на лекции знания, хранения примитивных и ссылочных типов данных;
- Получить практический навык создания функций по описанию;
- Попрактиковаться в написании простых функций, манипулирующих ссылочными данными.

Б.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучиает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	6-18	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (4 вопроса, по минуте на ответ)
Рассмотрение ДЗ лекции	10	19-23	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	24	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	10	25-28	Сравнить насколько разные могут быть прочтения одного и того же технического задания - одна функция для двух значений, возврат значений, возврат индекса, объявление исходного массива внутри функции поиска и др);
Задание 2	10	29-32	Корректная манипуляция индексами, как следствие, сокращение числа возможных проходов по массиву и ускорение работы приложени
Перерыв (если нужен)	5	33	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)

Что происходит	Время	Слайды	Описание
Задание 3	20	34-37	Формирование алгоритмического мышления при решении задач с описанием верхнего уровня
Задание 4	15	38-41	Понимание внутренней механики работы фреймворка коллекций, повышение уровня абстракции написанного кода
Задание 5 (необязат)	20	42-44	Описание базовых алгоритмов манипуляции данными с применением вспомогательных массивов
Домашнее задание	5	45-46	Объясните домашнее задание, подведите итоги урока
Рефлексия	10	47-48	Преподаватель запрашивает обратную связь
Длительность	125		

Б.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Презентует название курса (первый раз) и семинара;
  - Рассказывает немного о себе;
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5-7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
  - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
  - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Магическое число – это: (1)
     (a) числовая константа без пояснений;
     (b) число, помогающее в вычислениях;
     (c) числовая константа, присваиваемая при объявлении переменной.
  2. Какое значение будет содержаться в переменной a после выполнения строки `int a = 10.0f/3.0f;` (3)
  3. Сколько будет создано одномерных массивов при инициализации массива 3x3x3? (13)
  4. `2 + 2 * 2 == 2 << 2 >> 1`? (false? 6 != 4)

Рассмотрение ДЗ

- **Цель этапа:** Пояснить неочевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.

- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
  - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;
  - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
  - Написать метод «Шифр Цезаря», с булевым параметром зашифрования/расшифрования, и числовым ключом;

    Вариант решения

Листинг 106: Шифр Цезаря

```
private static String caesar(String in, int key, boolean encrypt) {
   if (in == null || in.isEmpty())
      return null;

   final int len = in.length();
   char[] out = new char[len];
   for (int i = 0; i < len; ++i) {
      out[i] = (char) (in.charAt(i) + ((encrypt) ? key : -key));
   }
   return new String(out);
}
```

  - Написать метод, принимающий на вход массив чисел и параметр `n`. Метод должен осуществить циклический (последний элемент при сдвиге становится первым) сдвиг всех элементов массива на n позиций;

    Вариант решения

Листинг 107: Сдвиговый метод

```
private static void shifter(int[] a, int n) {
   n %= a.length;
   int shift = a.length + n;
   shift %= a.length;

   for (int i = 0; i < shift; i++) {
      int temp = a[a.length - 1];
      System.arraycopy(a, 0, a, 1, a.length - 1);
      a[0] = temp;
   }
}
```

  - Написать метод, которому можно передать в качестве аргумента массив, состоящий строго из единиц и нулей (целые числа типа `int`). Метод должен заменить единицы в массиве на нули, а нули на единицы и не содержать ветвлений. Написать как можно больше вариантов метода

    Вариант решения

Листинг 108: Инверсия

```
 private static void change(int[] a) {
   for (int i = 0; i < a.length; i++) {
```

```
3        a[i] = 1 - a[i];
4  //       a[i] = (a[i] - 1) * -1;
5  //       a[i] = (a[i] + 1) % 2;
6     }
7  }
```

Вопросы и ответы

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
  - Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
  - Посоветовать литературу на тему заданного вопроса;
  - Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Базовая манипуляция данными внутри массива.
- **Тайминг** 10-15 минут.
- **Действия преподавателя**
  - Первые пять минут уклоняться от ответов на уточняющие вопросы
  - Пояснить студентам ценность опыта (сравнить насколько разные могут быть прочтения одного и того же технического задания - одна функция для двух значений, возврат значений, возврат индекса, объявление исходного массива внутри функции поиска и др);
  - Выдать задание группам студентов, показать где именно следует искать терминал ОС;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**:
  - Задать одномерный массив. Написать методы поиска в нём минимального и максимального элемента;

    Вариант решения

Листинг 109: Поиск минимального значения

```
1  private static int findMin(int[] a) { // returns the minimum value
2     int min = a[0];
3     for (int i = 1; i < a.length; i++) {
4        if (a[i] < min) {
5           min = a[i];
6        }
```

```
    }
    return min;
}
```

$*_1$ Привести функции к корректному виду и дополнительно написать ещё две функции так, чтобы получились (четыре) функции поиска минимального и максимального как значения, так и индекса.
Вариант решения

Листинг 110: Поиск индекса максимального значения

```
private static int findMax(int[] a) { // returns the maximum index
    int max = 0;
    for (int i = 1; i < a.length; i++) {
        if (a[i] > a[max])
        max = i;
    }
    return max;
  }
```

Задание 2

- **Ценность этапа** Оптимизация сложности алгоритмов при работе с многомерными массивами.
- **Тайминг** 10-15 минут.
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (корректная манипуляция индексами, как следствие, сокращение числа возможных проходов по массиву и ускорение работы приложения);
  - Пояснить студентам в каком виде выполнять и сдавать задания;
  - Выдать задание группам студентов, показать где и как скачивать необходимый инструментарий, если он ещё не установлен;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**
  - Создать квадратный целочисленный массив (количество строк и столбцов одинаковое), заполнить его диагональные элементы единицами, используя цикл(ы)
    Вариант решения

Листинг 111: Заполнение диагональных элементов

```
private static void fillDiagonal(int[][] a) {
    for (int i = 0; i < a.length; i++) {
        a[i][i] = 1;
        a[i][a.length - 1 - i] = 1;
    }
}
```

$*_1$ дописать функцию вывода двумерного массива в консоль
Вариант решения

Листинг 112: Вывод массива в терминал

```
private static void printTwoDimArray(int[][] a) {
    for (int i = 0; i < a.length; i++) {
        System.out.println(Arrays.toString(a[i]));
    }
}
```

Задание 3

- **Ценность этапа** Формирование алгоритмического мышления при решении задач с описанием верхнего уровня.
- **Тайминг** 15-20 минут
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (ТЗ довольно редко бывают чёткими и никогда не говорят программисту, что именно нужно написать);
  - Выдать задание группам студентов;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдать группе задание «со звёздочкой».
- **Задания**
  - Написать метод, в который передается не пустой одномерный целочисленный массив, метод должен вернуть true если в массиве есть место, в котором сумма левой и правой части массива равны. Примеры:

    `checkBalance([1, 1, 1, || 2, 1])` $\rightarrow$ `true`,
    `checkBalance([2, 1, 1, 2, 1])` $\rightarrow$ `false`,
    `checkBalance([10, || 1, 2, 3, 4])` $\rightarrow$ `true`.

    Абстрактная граница показана символами ||, эти символы в массив не входят.

    Вариант решения

Листинг 113: Вариант со сложностью $O(n^2)$

```
private static boolean checkBalance(int[] a) {
    int left = 0;
    for (int i = 0; i < a.length - 1; i++) {
        left += a[i];
        int right = 0;
        for (int j = i + 1; j < a.length; j++) {
            right += a[j];
        }
        if (left == right) return true;
    }
    return false;
}
```

Листинг 114: Вариант со сложностью $O(2n)$

```
private static boolean checkBalance2(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++) {
        sum += a[i];
    }
```

```
6       if (sum % 2 != 0) return false;
7       int left = 0;
8       for (int i = 0; i < a.length; i++) {
9           left += a[i];
10          sum -= a[i];
11          if (left == sum) return true;
12      }
13      return false;
14  }
```

$*_1$ написать этот же метод таким образом, чтобы в нём использовался только один цикл.
Вариант решения

Листинг 115: * Вариант со сложностью $O(n)$

```
1   private static boolean checkBalance3(int[] a) {
2       int lbound = 0;
3       int rbound = a.length - 1;
4       int left = 0;
5       int right = 0;
6       while (lbound <= rbound) {
7           if (left > right)
8               right += a[rbound--];
9           else
10              left += a[lbound++];
11      }
12      return left == right;
13  }
```

Задание 4

- **Ценность этапа** Понимание внутренней механики работы фреймворка коллекций, повышение уровня абстракции написанного кода.
- **Тайминг** 15-20 минут
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (написание собственных функций, реализующих алгоритмы часто помогает в ситуациях, когда задача не решается стандартными средствами)
  - Выдать задание группам студентов
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдать группе задание «со звёздочкой».
  - Если нужно, через 7 минут после старта, дать подсказку для первой части задания (сигнатура функции должна содержать не только передаваемый массив, но и его текущее заполнение, которое нужно отслеживать самостоятельно)
- **Задания**
  - Написать функцию добавления элемента в конец массива таким образом, чтобы она расширяла массив при необходимости.
    Здесь нет смысла показывать не лучшее, но самое популярное решение, поэтому можно продемонстрировать сразу вариант решения «со звёздочкой».

  $*_1$ Функция должна возвращать ссылку на вновь созданный внутри себя массив, а не

использовать глобальный
Вариант решения

Листинг 116: * Вариант без глобального массива

```
int[] add(int[] arr, int current, int value) {
   if (current == arr.length) {
      int[] temp = new int[arr.length * 2];
      System.arraycopy(arr, 0, temp, 0, arr.length);
      arr = temp;
   }
   arr[current++] = value;
   return arr;
}

// main
int[] array = {1,2};
int size = 2;
System.out.println(size + " = " + Arrays.toString(array));
array = add(array, size++, 6);
System.out.println(size + " = " + Arrays.toString(array));
array = add(array, size++, 6);
```

Задание 5 (необязательное)

- **Ценность этапа** Описание базовых алгоритмов манипуляции данными с применением вспомогательных массивов.
- **Тайминг** 15-20 минут
- **Действия преподавателя**
  - Объяснить студентам, в чём заключается алгоритм сортировки подсчётом. Для сортировки подсчётом алгоритм совершает проход по исходному массиву, подсчитывая количество повторений каждого значения, и записывая эту информацию в промежуточный (частотный) массив. Вторым шагом алгоритма совершается обход вспомогательного массива и запись нужного количества значений по возрастанию в исходный массив. Сложность сортировки $O(2n)$. Например:

$$x[2,1,0,4,3,0,0,1,2] \rightarrow t[3(x_0),2(x_1),2(x_2),1(x_3),1(x_4)] \rightarrow x[0,0,0,1,1,2,2,3,4]$$

  - Выдать задание группам студентов
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдать группе задание «со звёздочкой».
- **Задания**
  - Написать метод, осуществляющий сортировку одномерного массива подсчётом. Важное ограничение состоит в том, что для этой сортировки диапазон значений исходного массива должен находиться в разумных пределах, например, не более 1000.
  Вариант решения

Листинг 117: Pigeonhole sort

```
void pigeon(int[] arr) {
   final int min = getMin(arr);
```

```
3       final int max = getMax(arr);
4       int[] freq = new int[max - min + 1];
5       for (int i = 0; i < arr.length; i++)
6           freq[arr[i] - min]++;
7
8       int arrIndex = 0;
9       for (int i = 0; i < freq.length; i++)
10          for (int elems = freq[i]; elems > 0; elems--)
11              arr[arrIndex++] = i + min;
12  }
```

Домашнее задание

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки
  - Объяснить где взять проект заготовки для дз
- **Задания**

5-25 мин Решить все задания (в том числе «со звёздочкой»), если они не были решены на семинаре, без ограничений по времени;
Все варианты решения приведены в тексте семинара выше

5-10 мин Написать метод, возвращающий количество чётных элементов массива.

countEvens([2, 1, 2, 3, 4]) → 3
countEvens([2, 2, 0]) → 3
countEvens([1, 3, 5]) → 0

Листинг 118: CountEvens.java

```
1   int countEvens(int[] arr) {
2       int counter = 0;
3       for (int i = 0; i < arr.length; ++i) {
4           if (arr[i] % 2 == 0) {
5               counter++;
6           }
7       }
8       return counter;
9   }
```

10 мин Написать функцию, возвращающую разницу между самым большим и самым маленьким элементами переданного не пустого массива.

Листинг 119: Spread.java

```
1   int spread(int[] arr) {
2       int min = arr[0];
3       int max = arr[0];
4       for (int i = 1; i < arr.length; ++i) {
5           if (arr[i] < min) min = arr[i];
```

```
6         if (arr[i] > max) max = arr[i];
7      }
8      return max - min;
9   }
```

10 мин Написать функцию, возвращающую истину, если в переданном массиве есть два соседних элемента, с нулевым значением.

Листинг 120: Zero2.java

```
1  boolean zero2(int[] arr) {
2     for (int i = 0; i < arr.length - 1; ++i) {
3        if (arr[i] == 0 && arr[i + 1] == 0)
4        return true;
5     }
6     return false;
7  }
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

B. Семинар: классы и объекты

B.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

B.2. Цели семинара

- Закрепить полученные на лекции знания об объектах, наследовании и полиморфизме;
- Получить практический навык создания классов по описанию;
- Дополнительно рассмотреть использование свойств статичности сущностей, неочевидные случаи несоблюдения инкапсуляции;
- Попрактиковаться в написании простых классов и методов, манипулирующих ссылочными данными.

B.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучиает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	10	6-24	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (6 вопросов, по минуте на ответ)
Рассмотрение ДЗ лекции	10	25-30	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	31	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	5	32-35	Создание класса и объекта.
Задание 2	10	36-40	Манипуляция информацией об объекте
Перерыв (если нужен)	5	41	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 3	20	42-46	Создание и манипуляция множествами объектов
Задание 4	15	47-51	Манипуляции данными по условию, «массовое обслуживание»
Задание 5 (необязат)	15	52-54	Соблюдение атомарности методов, независимость методов от окружения

Что происходит	Время	Слайды	Описание
Домашнее задание	5	55-56	Объясните домашнее задание, подведите итоги урока
Рефлексия	10	57-58	Преподаватель запрашивает обратную связь
Длительность	120		

B.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5-7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
  - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
  - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Какое свойство добавляет ключевое слово static полю или методу? (2)
     (a) неизменяемость;
     (b) принадлежность классу;
     (c) принадлежность приложению.
  2. Что быстрее, стек или куча? (1)
     (a) куча;
     (b) стек;
     (c) одинаково.
  3. Для инициализации нового объекта абсолютно идентичными значениями свойств переданного объекта используется (3)
     (a) пустой конструктор;
     (b) конструктор по-умолчанию;
     (c) конструктор копирования.
  4. Инкапсуляция – это (2)
     (a) архивирование проекта
     (b) сокрытие информации о классе
     (c) создание микросервисной архитектуры
  5. Наследуются от Object (3)
     (a) строки
     (b) потоки ввода-вывода
     (c) и то, и другое
  6. Является ли перегрузка полиморфизмом (2)
     (a) да, это истинный полиморфизм

(b) да, это часть истинного полиморфизма
(c) нет, это не полиморфизм

Рассмотрение ДЗ

- **Цель этапа:** Пояснить не очевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
  - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;
  - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
  - Написать класс кота так, чтобы каждому объекту кота присваивался личный порядковый целочисленный номер;
    Вариант решения в приложении B.4
  - Написать классы кота и собаки, наследники животного. У всех есть три действия: бежать, плыть, прыгать. Действия принимают размер препятствия и возвращают булев результат. Три ограничения: высота прыжка, расстояние, которое животное может пробежать, расстояние, которое животное может проплыть. Следует учесть, что коты не любят воду.
    Вариант решения в приложении B.4
  - Добавить механизм, создающий 25% разброс значений каждого ограничения для каждого объекта.
    Вариант решения

Листинг 121: Конструктор с вариативностью

```
Animal(String type, String name, float maxJump, float maxRun, float maxSwim) {
    float jumpDiff = random.nextFloat() * maxJump - (maxJump / 2);
    float runDiff = random.nextFloat() * maxRun - (maxRun / 2);
    float swimDiff = random.nextFloat() * maxSwim - (maxSwim / 2);

    this.type = type;
    this.name = name;
    this.maxJump = maxJump + jumpDiff;
    this.maxRun = maxRun + runDiff;
    this.maxSwim = maxSwim + swimDiff;
}
```

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;

- Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
- Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
- Посоветовать литературу на тему заданного вопроса;
- Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Создание класса и объекта.
- **Тайминг** 5 минут.
- **Действия преподавателя**
  - Первые пять минут уклоняться от ответов на уточняющие вопросы
  - Выдать задание студентам;
- **Задания**:
  - Создать класс ”Сотрудник”с полями: ФИО, должность, телефон, зарплата, возраст; **Вариант исполнения класса в приложении В.4**

Листинг 122: Создание объекта класса

```
1 Employee employeeIvan = new Employee("Ivan", "Igorevich",
2       "Ovchinnikov", "8(495)000-11-22",
3       "developer", 50000, 1985);
```

Задание 2

- **Ценность этапа** Манипуляция информацией об объекте.
- **Тайминг** 10 минут.
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (отказ от вывода информации в терминал из сторонних объектов);
  - Пояснить студентам в каком виде выполнять и сдавать задания;
  - Выдать задание группам студентов;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**
  - Написать функцию выводящую всю доступную информацию об объекте **Вариант решения**

Листинг 123: Вывод информации об объекте в консоль

```
1   public void info() {
2      System.out.println("Employee{" +
3            "name='" + name + '\'' +
4            ", midName='" + midName + '\'' +
5            ", surName='" + surName + '\'' +
6            ", position='" + position + '\'' +
```

```
7               ", phone='" + phone + '\'' +
8               ", salary=" + salary +
9               ", age=" + getAge() +
10              '}');
11      }
```

$*_1$ таким образом, чтобы функция возвращала значение;
Вариант решения

Листинг 124: Возврат информации об объекте

```
1   @Override
2   public String toString() {
3       return "Employee{" +
4               "name='" + name + '\'' +
5               ", midName='" + midName + '\'' +
6               ", surName='" + surName + '\'' +
7               ", position='" + position + '\'' +
8               ", phone='" + phone + '\'' +
9               ", salary=" + salary +
10              ", age=" + getAge() +
11              '}';
12  }
```

$*_2$ с использованием форматирования строк.
Вариант решения

Листинг 125: Форматированная информация об объекте

```
1   @Override
2   public String toString() {
3       return String.format("Employee{" +
4               "name='%s', midName='%s', surName='%s'" +
5               ", position='%s', phone='%s'" +
6               ", salary=%d, age=%d}'",
7               name, midName, surName, position, phone, salary, getAge());
8   }
```

Задание 3

- **Ценность этапа** Создание и манипуляция множествами объектов.
- **Тайминг** 20-25 минут.
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (дополнительно напомнить, что классы создают для программы новый тип данных, а значит объектами можно пользоваться также, как и заранее созданными);
  - Пояснить студентам в каком виде выполнять и сдавать задания;
  - Выдать задание группам студентов;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**

– Создать массив из 5 сотрудников
Вариант решения

Листинг 126: Простой массив сотрудников

```
Employee ivan = new Employee("Ivan", "Igorevich",
      "Ovchinnikov", "8(495)000-11-22",
      "developer", 50000, 1985);
Employee andrey = new Employee("Andrey", "Viktorovich",
      "Bezrukov", "8(495)111-22-33",
      "fitter", 52000, 1973);
Employee evgeniy = new Employee("Evgeniy", "Viktorovich",
      "Delfinov", "8(495)222-33-44",
      "project manager", 40000, 1963);
Employee natalia = new Employee("Natalia", "Pavlovna",
      "Keks", "8(495)333-44-55",
      "senior developer", 90000, 1990);
Employee tatiana = new Employee("Tatiana", "Sergeevna",
      "Krasotkina", "8(495)444-55-66",
      "accountant", 50000, 1983);

Employee[] company = new Employee[5];
company[0] = ivan;
company[1] = andrey;
company[2] = evgeniy;
company[3] = natalia;
company[4] = tatiana;
```

$*_1$ массив должен быть сразу инициализирован;
Вариант решения

Листинг 127: Инициализированный массив

```
Employee[] employees = { ivan, andrey, evgeniy, natalia, tatiana };
```

$*_2$ массив должен быть сразу инициализирован и не должно быть создано дополнительных переменных.
Вариант решения

Листинг 128: Без дополнительных переменных

```
Employee[] employees = {
      new Employee("Ivan", "Igorevich",
            "Ovchinnikov", "8(495)000-11-22",
            "developer", 50000, 1985),
      new Employee("Andrey", "Viktorovich",
            "Bezrukov", "8(495)111-22-33",
            "fitter", 52000, 1973),
      new Employee("Evgeniy", "Viktorovich",
            "Delfinov", "8(495)222-33-44",
            "project manager", 40000, 1963),
      new Employee("Natalia", "Pavlovna",
            "Keks", "8(495)333-44-55",
```

```
13              "senior developer", 90000, 1990),
14        new Employee("Tatiana", "Sergeevna",
15              "Krasotkina", "8(495)444-55-66",
16              "accountant", 50000, 1983)
17  };
```

Задание 4

- **Ценность этапа** Манипуляция информацией об объекте.
- **Тайминг** 15-20 минут.
- **Действия преподавателя**
  - Пояснить студентам ценность этого опыта (часто возникают ситуации, когда необходимо осуществить фильтрацию данных или манипулировать только частью объектов. дополнительно указать на то, что часто методы, манипулирующие множеством объектов описывают внутри классов, но это не совсем верный подход с точки зрения архитектуры, и под такие методы желательно создавать отдельный класс, например, `EmployeeWorker`);
  - Пояснить студентам в каком виде выполнять и сдавать задания;
  - Выдать задание группам студентов;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**
  - Создать метод, повышающий зарплату всем сотрудникам старше 45 лет на 5000. Метод должен принимать в качестве параметра массив сотрудников.
    Вариант решения

Листинг 129: Метод повышения зарплаты

```
1  // Employee
2  public void increaseSalary(int amount) {
3     this.salary += amount;
4  }
5  // Main
6  private static void increaser(Employee[] emp) {
7     for (int i = 0; i < emp.length; i++) {
8        if (emp[i].getAge() > 45) {
9           emp[i].increaseSalary(5000);
10       }
11    }
12 }
13 // main
14 for (int i = 0; i < employees.length; i++) {
15    increaser(employees);
16 }
```

$*_1$ Написать тот же метод, но возраст и размер повышения должны быть параметрами метода.
Вариант решения

Листинг 130: Параметризированный метод повышения

```
// Main
private static void increaser(Employee[] emp, int age, int increment) {
   for (int i = 0; i < emp.length; i++) {
      if (emp[i].getAge() > age) {
         emp[i].increaseSalary(increment);
      }
   }
}
// main
for (int i = 0; i < employees.length; i++) {
   increaser(employees, 45, 5000);
}
```

$*_2$ Написать тот же метод в качестве статического в классе сотрудника.
Вариант решения

Листинг 131: Статический метод и использование

```
// Employee
public static void increaser(Employee[] emp, int age, int increment) {
   for (int i = 0; i < emp.length; i++) {
      if (emp[i].getAge() > age) {
         emp[i].increaseSalary(increment);
      }
   }
}
// main
for (int i = 0; i < employees.length; i++) {
   Employee.increaser(employees, 45, 5000);
}
```

Задание 5

- **Ценность этапа** Соблюдение атомарности методов, независимость методов, манипулирующих данными от окружения.
- **Тайминг** 10-15 минут.
- **Действия преподавателя**
  - Обратить внимание на то, что многие студенты пытаются объединить методы с похожими алгоритмами, но это неверный подход, нужно соблюдать атомарность методов. Важно, чтобы результаты работы не печатались в консоль, а возвращались из методов, чтобы код был переносимым;
  - Выдать задание группам студентов;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задания**
  - Написать методы (принимающие на вход массив сотрудников), вычисляющие средний возраст и среднюю зарплату сотрудников, вывести результаты работы в консоль.
    Вариант решения

Листинг 132: Методы подсчёта средних

```
private static float averageSalary(Employee[] emp) {
    float result = 0;
    for (int i = 0; i < emp.length; i++)
        result += emp[i].getSalary();

    return result / emp.length;
}

private static float averageAge(Employee[] emp){
    float result = 0;
    for (int i = 0; i < emp.length; i++)
        result += emp[i].getAge();

    return result / emp.length;
}
```

Домашнее задание

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  * Пояснить студентам в каком виде выполнять и сдавать задания
  * Уточнить кто будет проверять работы (преподаватель или ревьювер)
  * Объяснить к кому обращаться за помощью и где искать подсказки
  * Объяснить где взять проект заготовки для дз
- **Задания**

5-25 мин Решить все задания (в том числе «со звёздочкой»), если они не были решены на семинаре, без ограничений по времени;
Все варианты решения приведены в тексте семинара выше

5-10 мин 1. Написать прототип компаратора - метод внутри класса сотрудника, сравнивающий две даты, представленные в виде трёх чисел гггг-мм-дд, без использования условного оператора.

Листинг 133: Сравнение возраста

```
//Employee
int bMonth;
int bDay;
/**
 * returns
 * negative integer if this object birthdate is less (earlier),
    than given (older)
 * positive integer if this object birthdate is more (later),
    than given (younger)
 * zero if this object is the same as given
 * */
public int compare(int dd, int mm, int yyyy) {
    //day = 0..30, 31 is binary 11111, ok to left shift month by 6
    //month = 0..11, 15 is binary 1111, ok to left shift year by 5
```

```
       more
13  int empl = bDay + (bMonth << 6) + (birth << 11);
14  int income = dd + (mm << 6) + (yyyy << 11);
15  return empl - income;
16 }
```

10-15 мин 2. Опишите класс руководителя, наследник от сотрудника. Перенесите статический метод повышения зарплаты в класс руководителя, модифицируйте метод таким образом, чтобы он мог поднять заработную плату всем, кроме руководителей. В основной программе создайте руководителя и поместите его в общий массив сотрудников. Повысьте зарплату всем сотрудникам и проследите, чтобы зарплата руководителя не повысилась.

Листинг 134: Менеджер - это тоже сотрудник

```
1  // Manager
2  package ru.gb.jcore;
3
4  public class Manager extends Employee {
5
6     public Manager(String name, String midName, String surName,
7                 String phone, String position, int salary, int
                    birth) {
8        super(name, midName, surName, phone, position, salary,
              birth);
9     }
10    public static void increaser(Employee[] emp, int age, int
            increment) {
11       for (int i = 0; i < emp.length; i++) {
12          if (emp[i].getAge() > age) {
13             if (!(emp[i] instanceof Manager))
14                emp[i].increaseSalary(increment);
15          }
16       }
17    }
18 }
```

Листинг 135: Более верное повышение зарплаты

```
1  // main
2  Employee[] employees = { ivan, andrey, evgeniy /*new
       Manager(...)*/, natalia, tatiana };
3  for (int i = 0; i < employees.length; i++) {
4     Manager.increaser(employees, 45, 5000);
5  }
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать

- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

Приложения

Домашнее задание 1

Листинг 136: Кот

```
package ru.gb.jcore;

public class Cat {
    private static final int CURRENT_YEAR = 2022;
    private static int id = 0;

    private String name;
    private String color;
    private int birthYear;
    public int uid;

    Cat (String name, String color, int age) {
        setBirth(age);
        this.name = name;
        this.color = color;
        this.uid = ++id;
    }
    private void setBirth(int age) {
        this.birthYear = CURRENT_YEAR - age;
    }
    public int getUid() {
        return uid;
    }
    public String getColor() {
        return color;
    }
    public int getAge() {
        return CURRENT_YEAR - birthYear;
    }
    public String getName() {
        return name;
    }
}
```

Домашнее задание 2

Листинг 137: Общий класс животного

```
package ru.gb.jcore.marathon;

import java.util.Random;

```

```
public abstract class Animal {

    static final int SWIM_FAIL = 0;
    static final int SWIM_OK = 1;
    static final int SWIM_WTF = -1;

    private String type;
    private String name;
    private float maxRun;
    private float maxSwim;
    private float maxJump;
    private final Random random = new Random();

    Animal(String type, String name, float maxJump, float maxRun, float maxSwim)
        {
        this.type = type;
        this.name = name;
        this.maxJump = maxJump;
        this.maxRun = maxRun;
        this.maxSwim = maxSwim;
    }

    String getName() {
        return this.name;
    }
    String getType() {
        return this.type;
    }
    float getMaxRun() {
        return this.maxRun;
    }
    float getMaxSwim() {
        return this.maxSwim;
    }
    float getMaxJump() {
        return this.maxJump;
    }
    protected boolean run(float distance) {
        return (distance <= maxRun);
    }
    protected int swim(float distance) {
        return (distance <= maxSwim) ? SWIM_OK : SWIM_FAIL;
    }
    protected boolean jump(float height) {
        return (height <= maxJump);
    }
}
```

Листинг 138: Кот

```
package ru.gb.jcore.marathon;

public class Cat extends Animal {

   Cat(String name) {
      super("Cat", name, 2, 200, 1);
   }

   @Override
   protected int swim(float distance) {
      return Animal.SWIM_WTF;
   }
}
```

Листинг 139: Собака

```
package ru.gb.jcore.marathon;

class Dog extends Animal {

   Dog(String name, float maxJump, float maxRun, float maxSwim) {
      super("Dog", name, maxJump, maxRun, maxSwim);
   }

}
```

Листинг 140: Марафон

```
package ru.gb.jcore.marathon;

public class Marathon {
   public static void main(String[] args) {

      Cat c = new Cat("Barseek");
      Cat c0 = new Cat("Moorzeek");
      Dog d = new Dog("Toozeek", 0.5f, 500, 10);
      Dog d0 = new Dog("Shaareek", 0.5f, 500, 10);

      Animal[] arr = {c, c0, d, d0};
      float toJump = 1.5f;
      float toRun = 350;
      float toSwim = 5;

      for (int i = 0; i < arr.length; i++) {
         String nameString = arr[i].getType() + " " + arr[i].getName() + " can
             ";

         String eventName = String.format("jump max %.2fm. Tries to jump ",
             arr[i].getMaxJump());
         String eventResult = (arr[i].jump(toJump)) ? "succeed" : "fails";
```

```
21          System.out.println(nameString + eventName + toJump + "m and " +
               eventResult);
22
23          eventName = String.format("run max %.2fm. Tries to run ",
               arr[i].getMaxRun());
24          eventResult = arr[i].run(toRun) ? "succeed" : "fails";
25          System.out.println(nameString + eventName + toRun + "m and " +
               eventResult);
26
27          int swimResult = arr[i].swim(toSwim);
28          eventName = String.format("swim max %.2fm. Tries to swim ",
               arr[i].getMaxSwim());
29          eventResult = (swimResult == Animal.SWIM_OK) ? "succeed" : "fails";
30          if (swimResult == Animal.SWIM_WTF)
31             eventResult = "too scared to enter the water";
32          System.out.println(nameString + eventName + toSwim + "m and " +
               eventResult);
33       }
34    }
35 }
```

Практическое задание 1

Листинг 141: Сотрудник

```
1  package ru.gb.jcore;
2
3  public class Employee {
4     private static final int CURRENT_YEAR = 2022;
5     String name;
6     String midName;
7     String surName;
8     String position;
9     String phone;
10    int salary;
11    int birth;
12
13    public Employee(String name, String midName, String surName,
14                String phone, String position, int salary, int birth) {
15       this.name = name;
16       this.midName = midName;
17       this.surName = surName;
18       this.position = position;
19       this.phone = phone;
20       this.salary = salary;
21       this.birth = birth;
22    }
23
24    public String getName() {
```

```
25          return name;
26      }
27  
28      public String getMidName() {
29          return midName;
30      }
31  
32      public String getSurName() {
33          return surName;
34      }
35  
36      public String getPosition() {
37          return position;
38      }
39  
40      public void setPosition(String position) {
41          this.position = position;
42      }
43  
44      public String getPhone() {
45          return phone;
46      }
47  
48      public void setPhone(String phone) {
49          this.phone = phone;
50      }
51  
52      public int getSalary() {
53          return salary;
54      }
55  
56      public void setSalary(int salary) {
57          this.salary = salary;
58      }
59  
60      public int getAge() {
61          return CURRENT_YEAR - birth;
62      }
63  
64  }
```

Г. Семинар: обработка исключений

Г.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

Г.2. Цели семинара

- Закрепить полученные на лекции знания об исключениях;
- Попрактиковаться в написании собственных классов исключений;
- Создание, выбрасывание и обработка исключений;
- Обработка исключений в стиле «до захвата ресурса»;
- Проброс исключений выше по стеку.

Г.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучиает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	6-18	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (6 вопросов, по минуте на ответ)
Рассмотрение ДЗ лекции	10	19-23	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	24	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	30	25-35	Сквозное задание, состоящее из объяснений в 6 пунктах, обязательных к исполнению. Всё задание выдаётся сразу целиком и является неделимым.
Перерыв (если нужен)	5	36	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 2	40	37-49	Сквозное задание, состоящее из объяснений в 6 пунктах, обязательных к исполнению. Всё задание выдаётся сразу целиком и является неделимым.
Домашнее задание	5	50-51	Объясните домашнее задание, подведите итоги урока
Рефлексия	10	52-53	Преподаватель запрашивает обратную связь

Что происходит	Время	Слайды	Описание
Длительность	120		

Г.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5-7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
  - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
  - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Перечисление – это: 2
     (a) массив
     (b) класс
     (c) объект
  2. Инкапсуляция с использованием внутренних классов: 2
     (a) остаётся неизменной
     (b) увеличивается
     (c) уменьшается
  3. Обработчик исключений – это объект, работающий 1
     (a) в специальном потоке
     (b) в специальном ресурсе
     (c) в специальный момент
  4. Стектрейс - это 2
     (a) часть потока выполнения программы;
     (b) часть объекта исключения;
     (c) часть информации в окне отладчика.

Рассмотрение ДЗ

- **Цель этапа:** Пояснить не очевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
  - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;
  - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.

- **Домашнее задание из лекции:**
  - Напишите два наследника класса Exception: ошибка преобразования строки и ошибка преобразования столбца.
    Вариант решения

Листинг 142: Простые наследники

```
private static final class ColumnMismatchException extends
    RuntimeException {
   ColumnMismatchException(String message) {
      super("Columns exception: " + message);
   }
}

private static final class NumberIsNotNumberException extends
    RuntimeException {
   NumberIsNotNumberException(String message) {
      super("Not a number found: " + message);
   }
}
```

  - Разработайте исключения-наследники так, чтобы они информировали пользователя в формате ожидание/реальность.

Листинг 143: Составное сообщение

```
private static final class RowMismatchException extends
    RuntimeException {
   RowMismatchException(int expected, int current, String value) {
      super(String.format("Rows exception: expected %d rows. Received
          %d rows in '%s' string",
             expected, current, value));
   }
}
```

  - для проверки напишите программу, преобразующую квадратный массив целых чисел 5x5 в сумму чисел в этом массиве, при этом, программа должна выбросить исключение, если строк или столбцов в исходном массиве окажется не 5.
    Вариант решения представлен в приложении Г.4

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);

- Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
- Посоветовать литературу на тему заданного вопроса;
- Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Написание почти полноценной механики по краткому ТЗ.
- **Тайминг** 25-30 минут.
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций.
- **Задание**: Класс «Проверка логина и пароля».
  1. Создать статический метод который принимает на вход три параметра: `login`, `password` и `confirmPassword`.
  2. Длина `login` должна быть **меньше** 20 символов. Если `login` не соответствует этому требованию, необходимо выбросить `WrongLoginException`.
  3. Длина `password` должна быть **не меньше** 20 символов. Также `password` и `confirmPassword` должны быть равны. Если `password` не соответствует этим требованиям, необходимо выбросить `WrongPasswordException`.
  4. `WrongPasswordException` и `WrongLoginException` – пользовательские классы исключения с двумя конструкторами –- один по умолчанию, второй принимает параметры исключения (неверные данные) и возвращает пользователю в виде «ожидалось/фактически».
  5. В основном классе программы необходимо по-разному обработать исключения.
  6. Метод возвращает `true`, если значения верны или `false` в противном случае.

  Вариант исполнения класса в приложении Г.4

  Вариант маршрута решения задачи

Листинг 144: Описание класса исключения логина

```
1  public static class WrongLoginException extends RuntimeException {
2      private int currentLength;
3      public WrongLoginException(int currentLength) {
4          super();
5          this.currentLength = currentLength;
6      }
7
8      @Override
9      public String getMessage() {
10         return String.format("Your login is of incorrect length, expected <
               20, given %d.",
11                 currentLength);
12     }
13 }
```

Листинг 145: Формирование сообщения для класса исключения пароля

```
public static class WrongPasswordException extends RuntimeException {
    private int currentLength;
    private boolean matchConfirm;
    public WrongPasswordException(int currentLength, boolean matchConfirm) {
        super();
        this.currentLength = currentLength;
        this.matchConfirm = matchConfirm;
    }

    @Override
    public String getMessage() {
        boolean badlen = currentLength <= 20;
        return String.format("Your password is of %scorrect length%s %d.
            Password %smatch the confirmation.",
                ((badlen) ? "in" : ""),
                ((badlen) ? ", expected > 20, given" : ","),
                currentLength,
                (matchConfirm) ? "" : "doesn't ");
    }
}
```

Листинг 146: Создание тестовой среды

```
public static void main(String[] args) {
    String[][] credentials = {
            {"ivan", "1i2v3a4n5i6v7a8n91011", "1i2v3a4n5i6v7a8n91011"},
                //correct
            {"1i2v3a4n5i6v7a8n91011", "", ""}, //wrong login length
            {"ivan", "1i2v3a4n5i6v7a8n91011", "1i2v3a4n5"}, //confirm mismatch
            {"ivan", "1i2v3a4n5", "1i2v3a4n5"},//wrong password length
            {"ivan", "1i2v3a4n5", "1i"} //wrong password length and confirm
                mismatch
    };
    for (int i = 0; i < credentials.length; i++) {
        try {
            System.out.println(checkCredentials(credentials[i][0],
                credentials[i][1], credentials[i][2]));
        } catch (WrongLoginException e) {
            e.printStackTrace();
        } catch (WrongPasswordException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Листинг 147: Метод проверки

```
public static boolean checkCredentials(String login, String password,
    String confirmPassword) {
    boolean conf = password.equals(confirmPassword);
```

```
3       int llen = login.length();
4       int plen = password.length();
5       if (llen >= 20)
6           throw new WrongLoginException(llen);
7       else if (plen < 20 || !conf)
8           throw new WrongPasswordException(plen, conf);
9       else
10          return true;
11  }
```

Задание 2

- **Ценность этапа** Написание наброска пет-проекта, повторение информации об ООП, работа с исключениями.
- **Тайминг** 35-40 минут.
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций.
  - обратить внимание на порядок обработки исключений, повторная попытка купить товар может производиться только тогда, когда остальные параметры покупки уже проверены.
- **Задание**: Класс «Эмуляция интернет-магазина».
  1. Написать классы покупатель (ФИО, возраст, телефон), товар (название, цена) и заказ (объект покупатель, объект товар, целочисленное количество).
  2. Создать массив покупателей (инициализировать 2 элемента), массив товаров (инициализировать 5 элементов) и массив заказов (пустой на 5 элементов).
  3. Создать статический метод «совершить покупку» со строковыми параметрами, соответствующими полям объекта заказа. Метод должен вернуть объект заказа.
  4. Если в метод передан несуществующий покупатель – метод должен выбросить исключение `CustomerException`, если передан несуществующий товар, метод должен выбросить исключение `ProductException`, если было передано отрицательное или слишком больше значение количества (например, 100), метод должен выбросить исключение `AmountException`.
  5. Вызвать метод совершения покупки несколько раз таким образом, чтобы заполнить массив покупок возвращаемыми значениями. Обработать исключения следующим образом (в заданном порядке):
     - если был передан неверный товар – вывести в консоль сообщение об ошибке, не совершать данную покупку;
     - если было передано неверное количество – купить товар в количестве 1;
     - если был передан неверный пользователь – завершить работу приложения с исключением.
  6. Вывести в консоль итоговое количество совершённых покупок после выполнения основного кода приложения.

  Вариант исполнения класса в приложении Г.4
  Вариант маршрута решения задачи

Листинг 148: Создание базовых классов (идентично)

```
1   private static class Item {
```

```
    String name;
    int cost;

    public Item(String name, int cost) {
       this.name = name;
       this.cost = cost;
    }

    @Override
    public String toString() {
       return "Item{name='" + name + "', cost=" + cost + "}";
    }
}
```

Листинг 149: Создание пользовательских исключений

```
    public static class CustomerException extends RuntimeException {
       public CustomerException(String message) {
          super(message);
       }
    }
```

Листинг 150: Создание массивов

```
private final static Customer[] people = {
      new Customer("Ivan", 20, "+1-222-333-44-55"),
      new Customer("Petr", 30, "+2-333-444-55-66"),
};
private final static Item[] items = {
      new Item("Ball", 100),
      new Item("Sandwich", 1000),
      new Item("Table", 10000),
      new Item("Car", 100000),
      new Item("Rocket", 10000000)
};
private static Order[] orders = new Order[5];
```

Листинг 151: Описание тестовой среды

```
Object[][] info = {
      {people[0], items[0], 1}, //good
      {people[1], items[1], -1}, //bad amount -1
      {people[0], items[2], 150}, //bad amount >100
      {people[1], new Item("Flower", 10), 1}, //no item
      {new Customer("Fedor", 40, "+3-444-555-66-77"), items[3], 1}, //no
         customer
};
int capacity = 0;
int i = 0;
while (capacity != orders.length - 1 || i != info.length) {
```

```
11        try {
12            orders[capacity] = buy((Customer) info[i][0], (Item) info[i][1],
                  (int) info[i][2]);
13            capacity++;
14        } catch (ProductException e) {
15            e.printStackTrace();
16        } catch (AmountException e) {
17            orders[capacity++] = buy((Customer) info[i][0], (Item) info[i][1],
                  1);
18        } catch (CustomerException e) {
19            throw new RuntimeException(e);
20        } finally {
21            System.out.println("Orders made: " + capacity);
22        }
23        ++i;
24    }
```

Листинг 152: Написание и **отладка** основного метода

```
1  private static boolean isInArray(Object[] arr, Object o) {
2      for (int i = 0; i < arr.length; i++)
3          if (arr[i].equals(o)) return true;
4      return false;
5  }
6
7  public static Order buy(Customer who, Item what, int howMuch) {
8      if (!isInArray(people, who))
9          throw new CustomerException("Unknown customer: " + who);
10     if (!isInArray(items, what))
11         throw new ProductException("Unknown item: " + what);
12     if (howMuch < 0 || howMuch > 100)
13         throw new AmountException("Incorrect amount: " + howMuch);
14     return new Order(who, what, howMuch);
15 }
```

Домашнее задание

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки
  - Объяснить где взять проект заготовки для дз
- **Задания**

5-25 мин Выполнить все задания семинара, если они не были решены, без ограничений по времени;

Все варианты решения приведены в тексте семинара выше

15 мин 1. В класс покупателя добавить перечисление с гендерами, добавить в сотрудника

свойство «пол» со значением созданного перечисления. Добавить геттеры, сеттеры.

Листинг 153: Свойства сотрудника

```
enum Genders{MALE, FEMALE};

// ...
Genders gender;

public Employee(String name, String midName, String surName, String
    phone, String position, int salary, int birth, Genders gender) {
    // ...
    this.gender = gender;
}

public Genders getGender() {
    return gender;
}

public void setGender(Genders gender) {
    this.gender = gender;
}
```

20-25 мин 2. Добавить в основную программу перечисление с праздниками (нет праздника, Новый Год, 8 марта, 23 февраля), написать метод, принимающий массив сотрудников, поздравляющий всех сотрудников с Новым Годом, женщин с 8 марта, а мужчин с 23 февраля, если сегодня соответствующий день.

Листинг 154: Праздники

```
enum Parties{NONE, NEW_YEAR, MARCH_8, FEB_23}
private static final Parties today = Parties.NONE;

private static void celebrate(Employee[] emp) {
    for (int i = 0; i < emp.length; i++) {
        switch (today) {
            case NEW_YEAR:
                System.out.println(emp[i].name + ", happy New Year!");
                break;
            case FEB_23:
                if (emp[i].gender == Employee.Genders.MALE)
                    System.out.println(emp[i].name + ", happy February
                        23rd!");
                break;
            case MARCH_8:
                if (emp[i].gender == Employee.Genders.FEMALE)
                    System.out.println(emp[i].name + ", happy march 8th!");
                break;
            default:
                System.out.println(emp[i].name + ", celebrate this
                    morning!");
        }
```

```
21      }
22  }
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

Приложения

Домашнее задание 3

Листинг 155: Основная программа

```java
package ru.gb.jcore;

import java.util.Arrays;

public class Exceptional {
   private static final class ColumnMismatchException extends RuntimeException {
      ColumnMismatchException(String message) {
         super("Columns exception: " + message);
      }
   }

   private static final class NumberIsNotNumberException extends
       RuntimeException {
      NumberIsNotNumberException(String message) {
         super("Not a number found: " + message);
      }
   }

   private static final class RowMismatchException extends RuntimeException {
      RowMismatchException(int expected, int current, String value) {
         super(String.format("Rows exception: expected %d rows. Received %d
            rows in '%s' string",
               expected, current, value));
      }
   }

   private static final String CORRECT_STRING= "1 3 1 2\n2 3 2 2\n5 6 7 1\n3 3
       1 0";
   private static final String EXTRA_ROW_STRING= "1 3 1 2\n2 3 2 2\n5 6 7 1\n3
       3 1 0\n1 2 3 4";
   private static final String EXTRA_COL_STRING= "1 3 1 2 1\n2 3 2 2 1\n5 6 7 1
       1\n3 3 1 0 1";
   private static final String NO_ROW_STRING= "1 3 1 2\n2 3 2 2\n5 6 7 1";
   private static final String NO_COL_STRING= "1 3 1 2\n2 3 2 2\n5 6 7 1\n3 3
       1";
   private static final String HAS_CHAR_STRING= "1 3 1 2\n2 3 2 2\n5 6 7 1\n3 3
       1 A";

   private static final int MATRIX_ROWS= 4;
   private static final int MATRIX_COLS= 4;

   private static String[][] stringToMatrix(String value) {
      String[] rows = value.split("\n");
      if (rows.length !=MATRIX_ROWS)
```

```
38            throw new RowMismatchException(MATRIX_ROWS, rows.length, value);
39
40         String[][] result = new String[MATRIX_ROWS][];
41         for (int i = 0; i < result.length; i++) {
42            result[i] = rows[i].split(" ");
43            if (result[i].length !=MATRIX_COLS)
44               throw new ColumnMismatchException(result[i].length + ":\n" + value);
45         }
46         return result;
47      }
48
49      private static float calcMatrix(String[][] matrix) {
50         float result = 0;
51         int len = 0;
52         for (int i = 0; i < matrix.length; i++) {
53            for (int j = 0; j < matrix[i].length; j++) {
54               try {
55                  result += Integer.parseInt(matrix[i][j]);
56                  ++len;
57               } catch (NumberFormatException e) {
58                  throw new NumberIsNotNumberException(matrix[i][j]);
59               }
60            }
61         }
62         return result / len;
63      }
64
65      public static void main(String[] args) {
66         try {
67   //          final String[][] matrix = stringToMatrix(CORRECT_STRING);
68   //          final String[][] matrix = stringToMatrix(NO_ROW_STRING);
69   //          final String[][] matrix = stringToMatrix(NO_COL_STRING);
70            final String[][] matrix =stringToMatrix(HAS_CHAR_STRING);
71            System.out.println(Arrays.deepToString(matrix));
72            System.out.println("Half sum = " +calcMatrix(matrix));
73         } catch (NumberIsNotNumberException exceptionObjectName) {
74            System.out.println("A NumberFormatException is thrown: " +
                  exceptionObjectName.getMessage());
75         } catch (RowMismatchException | ColumnMismatchException
               superExceptionName) {
76            System.out.println("A RuntimeException successor is thrown: " +
                  superExceptionName.getMessage());
77         }
78      }
79   }
```

Практическое задание 1

Листинг 156: Логин

```java
package ru.gb.jcore;

public class SignInWorker {
   public static class WrongPasswordException extends RuntimeException {
      private int currentLength;
      private boolean matchConfirm;
      public WrongPasswordException(int currentLength, boolean matchConfirm) {
         super();
         this.currentLength = currentLength;
         this.matchConfirm = matchConfirm;
      }

      @Override
      public String getMessage() {
         boolean badlen = currentLength <= 20;
         return String.format("Your password is of %scorrect length%s %d.
            Password %smatch the confirmation.",
               ((badlen) ? "in" : ""),
               ((badlen) ? ", expected > 20, given" : ","),
               currentLength,
               (matchConfirm) ? "" : "doesn't ");
      }
   }

   public static class WrongLoginException extends RuntimeException {
      private int currentLength;
      public WrongLoginException(int currentLength) {
         super();
         this.currentLength = currentLength;
      }

      @Override
      public String getMessage() {
         return String.format("Your login is of incorrect length, expected <
            20, given %d.",
               currentLength);
      }
   }

   public static boolean checkCredentials(String login, String password, String
      confirmPassword) {
      boolean conf = password.equals(confirmPassword);
      int llen = login.length();
      int plen = password.length();
      if (llen >= 20)
         throw new WrongLoginException(llen);
      else if (plen < 20 || !conf)
         throw new WrongPasswordException(plen, conf);
      else
```

```
        return true;
    }

    public static void main(String[] args) {
        String[][] credentials = {
                {"ivan", "1i2v3a4n5i6v7a8n91011", "1i2v3a4n5i6v7a8n91011"},
                    //correct
                {"1i2v3a4n5i6v7a8n91011", "", ""}, //wrong login length
                {"ivan", "1i2v3a4n5i6v7a8n91011", "1i2v3a4n5"}, //confirm mismatch
                {"ivan", "1i2v3a4n5", "1i2v3a4n5"},//wrong password length
                {"ivan", "1i2v3a4n5", "1i"} //wrong password length and confirm
                    mismatch
        };
        for (int i = 0; i < credentials.length; i++) {
            try {
                System.out.println(checkCredentials(credentials[i][0],
                    credentials[i][1], credentials[i][2]));
            } catch (WrongLoginException e) {
                e.printStackTrace();
            } catch (WrongPasswordException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Практическое задание 2

Листинг 157: Магазин

```
package ru.gb.jcore;

public class Shop {
    /**
     * Вызватьметодсовершенияпокупкинесколькоразтакимобразом       ,
     * чтобызаполнитьмассивпокупоквозвращаемымизначениями      .
     * Обработатьисключенияследующимобразомв     ( заданномпорядке):
     * */
    private static class Customer {
        String name;
        int age;
        String phone;

        public Customer(String name, int age, String phone) {
            this.name = name;
            this.age = age;
            this.phone = phone;
        }

```

```
20        @Override
21        public String toString() {
22            return "Customer{name='" + name + '\'' +
23                    ", age=" + age + ", phone='" + phone + "'}";
24        }
25    }
26    private static class Item {
27        String name;
28        int cost;
29
30        public Item(String name, int cost) {
31            this.name = name;
32            this.cost = cost;
33        }
34
35        @Override
36        public String toString() {
37            return "Item{name='" + name + "', cost=" + cost + "}";
38        }
39    }
40
41    private static class Order {
42        Customer customer;
43        Item item;
44        int amount;
45
46        public Order(Customer customer, Item item, int amount) {
47            this.customer = customer;
48            this.item = item;
49            this.amount = amount;
50        }
51
52        @Override
53        public String toString() {
54            return "Order{customer=" + customer +
55                    ", item=" + item + ", amount=" + amount + "}";
56        }
57    }
58
59    public static class CustomerException extends RuntimeException {
60        public CustomerException(String message) { super(message); }
61    }
62    public static class ProductException extends RuntimeException {
63        public ProductException(String message) { super(message); }
64    }
65    public static class AmountException extends RuntimeException {
66        public AmountException(String message) { super(message); }
67    }
68
69    private final static Customer[] people = {
70            new Customer("Ivan", 20, "+1-222-333-44-55"),
```

```
71            new Customer("Petr", 30, "+2-333-444-55-66"),
72      };
73      private final static Item[] items = {
74            new Item("Ball", 100),
75            new Item("Sandwich", 1000),
76            new Item("Table", 10000),
77            new Item("Car", 100000),
78            new Item("Rocket", 10000000)
79      };
80      private static Order[] orders = new Order[5];
81
82      private static boolean isInArray(Object[] arr, Object o) {
83         for (int i = 0; i < arr.length; i++)
84            if (arr[i].equals(o)) return true;
85         return false;
86      }
87
88      public static Order buy(Customer who, Item what, int howMuch) {
89         if (!isInArray(people, who))
90            throw new CustomerException("Unknown customer: " + who);
91         if (!isInArray(items, what))
92            throw new ProductException("Unknown item: " + what);
93         if (howMuch < 0 || howMuch > 100)
94            throw new AmountException("Incorrect amount: " + howMuch);
95         return new Order(who, what, howMuch);
96      }
97
98      public static void main(String[] args) {
99         Object[][] info = {
100              {people[0], items[0], 1}, //good
101              {people[1], items[1], -1}, //bad amount -1
102              {people[0], items[2], 150}, //bad amount >100
103              {people[1], new Item("Flower", 10), 1}, //no item
104              {new Customer("Fedor", 40, "+3-444-555-66-77"), items[3], 1}, //no
                    customer
105        };
106        int capacity = 0;
107        int i = 0;
108        while (capacity != orders.length - 1 || i != info.length) {
109           try {
110              orders[capacity] = buy((Customer) info[i][0], (Item) info[i][1],
                    (int) info[i][2]);
111              capacity++;
112           } catch (ProductException e) {
113              e.printStackTrace();
114           } catch (AmountException e) {
115              orders[capacity++] = buy((Customer) info[i][0], (Item) info[i][1],
                    1);
116           } catch (CustomerException e) {
117              throw new RuntimeException(e);
118           } finally {
```

```
119                 System.out.println("Orders made: " + capacity);
120             }
121             ++i;
122         }
123     }
124 }
```

Д. Семинар: тонкости работы

Д.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

Д.2. Цели семинара

- Практика создания, отправки и принятия данных;
- сохранение и загрузка состояния программы между запусками;
- работа с большими текстами (поиск, замена, генерация);
- начало рассмотрения популярных пакетов ввода-вывода.

Д.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучивает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	6-18	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (6 вопросов, по минуте на ответ)
Рассмотрение ДЗ лекции	10	19-27	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	28	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	10	29-33	Сохранение состояния приложения между запусками
Задание 2	15	34-37	Загрузка состояния приложения при запуске
Перерыв (если нужен)	5	38	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 3	10	39-42	Работа с текстом (автоматизированный поиск и замена текста в файле или группе файлов)
Задание 4	15	43-46	Работа с файловой системой
Задание 5 (необязат)	20	47-49	Описание часто недостающих механик операционной системы по «массовой» работе с файлами
Домашнее задание	5	50-51	Объясните домашнее задание, подведите итоги урока

Что происходит	Время	Слайды	Описание
Рефлексия	10	52-53	Преподаватель запрашивает обратную связь
Длительность	120		

Д.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5-7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
  - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
  - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Что такое GPT? (2)
     (a) General partition trace;
     (b) GUID partition table;
     (c) Greater pass timing.
  2. Строки в языке Java – это: (3)
     (a) классы;
     (b) массивы;
     (c) объекты.
  3. Ссылка на местонахождение – это: (2)
     (a) URI;
     (b) URL;
     (c) URN.
  4. Возможно ли чтение совершенно случайного байта данных из потока, к которому подключен объект `BufferedInputStream`? (2)
     (a) да;
     (b) нет;
     (c) да, если это поток в программу из локального файла.

Рассмотрение ДЗ

- **Цель этапа:** Пояснить не очевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
  - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;

   - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
   - Создать пару-тройку текстовых файлов. Для упрощения (чтобы не разбираться с кодировками) внутри файлов следует писать текст только латинскими буквами.

     Вариант решения

     Задание не предполагало кода, но будет плюсом, если студенты создали файлы не вручную, а программным кодом. Для этого и последующих заданий понадобится набор констант и вспомогательный массив с названиями файлов

```
private static final Random rnd = new Random();
private static final int CHAR_BOUND_L = 65;
private static final int CHAR_BOUND_H = 90;
private static final int FILES_AMOUNT = 10;
private static final int WORDS_AMOUNT = 5;
private static final int WORD_LENGTH = 10;
private static final String WORD_TO_SEARCH = "geekbrains";

String[] fileNames = new String[FILES_AMOUNT];
for (int i = 0; i < fileNames.length; i++)
    fileNames[i] = "file_" + i + ".txt";
```

Файл записывается простым выходным потоком, в который (в приведённом ниже примере) записывается строка, сгенерированная методом. Из строки выделяется массив байтов методом `getBytes()`.

```
private static String generateSymbols(int amount) {
    StringBuilder sequence = new StringBuilder();
    for (int i = 0; i < amount; i++)
        sequence.append((char) (CHAR_BOUND_L + rnd.nextInt(CHAR_BOUND_H
            - CHAR_BOUND_L)));
    return sequence.toString();
}

private static void writeFileContents(String fileName, int length)
    throws IOException {
    FileOutputStream fos = new FileOutputStream(fileName);
    fos.write(generateSymbols(length).getBytes());
    fos.flush();
    fos.close();
}
```

Вызов методов обёрнут в `try...catch` и формирует сразу файлы как для второго задания (конкатенации), так и третьего (поиска).

```
try {
    //#1
    for (int i = 0; i < fileNames.length; i++)
        if (i < 2)
            writeFileContents(fileNames[i], 100);
        else
            writeFileContents(fileNames[i], WORDS_AMOUNT, WORD_LENGTH);
```

```
    System.out.println("First task results are in file_0 and file_1.");
}
catch (Exception ex) { throw new RuntimeException(ex); }
```

- Написать метод, осуществляющий конкатенацию (соединение) переданных ей в качестве параметров файлов (не особенно важно, в первый допишется второй или во второй первый, или файлы вовсе объединятся в какой-то третий);
  Вариант решения
  Метод конкатенации может также быть написан множеством способов, но важно, чтобы при буферизации (чаще всего решения содержат именно буферизованные варианты) не терялись символы, например, пробелы или переходы на новую строку. Также часто встречается дополнительная буферизация внутри программы, например, в строку, при помощи `StringBuilder`, что будет являться грубой ошибкой при конкатенации больших или бинарных файлов. Поэтому ниже представлен вариант решения с посимвольным чтением и записью в результирующий файл.

```
private static void concatenate(String file_in1, String file_in2,
    String file_out) throws IOException {
    FileOutputStream fos = new FileOutputStream(file_out);
    int ch;
    FileInputStream fis = new FileInputStream(file_in1);
    while ((ch = fis.read()) != -1)
        fos.write(ch);
    fis.close();

    fis = new FileInputStream(file_in2);
    while ((ch = fis.read()) != -1)
        fos.write(ch);
    fis.close();

    fos.flush();
    fos.close();
}
```

  Вызов метода можно записать в секцию `try...catch` предыдущего задания или написать новую.

```
try {
    //#2
    concatenate(fileNames[0], fileNames[1], "FILE_OUT.txt");
    System.out.println("Second task result is in FILE_OUT.");
}
catch (Exception ex) { throw new RuntimeException(ex); }
```

- Написать метод поиска слова внутри файла.
  Вариант решения
  Для поиска слова понадобится файл не со сплошными символами (хотя и в таком алгоритм поиска должен отработать нормально), а со словами, разделёнными пробелами. Для этого можно видоизменить метод генерации файла и написать перегруженный, который будет по некоторому псевдослучайному условию вставлять в файл через пробел либо случайно сгенерированную последовательность символов, либо слово, заранее определённое, как искомое.

```java
private static void writeFileContents(String fileName, int words, int
    length) throws IOException {
   FileOutputStream fos = new FileOutputStream(fileName);
   for (int i = 0; i < words; i++) {
      if(rnd.nextInt(WORDS_AMOUNT) == WORDS_AMOUNT / 2)
         fos.write(WORD_TO_SEARCH.getBytes());
      else
         fos.write(generateSymbols(length).getBytes());
      fos.write(' ');
   }
   fos.flush();
   fos.close();
}
```

Формально, можно выполнить работу следующим образом, используя Scanner, сравнивая слова «от пробела до пробела»

```java
private static boolean fileScanner(String fileName, String word)
    throws IOException {
   Scanner sc = new Scanner(new FileInputStream(fileName));
   word = word.toLowerCase(); // \n
   while (sc.hasNext()) {
      String line = sc.nextLine();
      line = line.toLowerCase();
      if (line.contains(word)) return true;
   }
   return false;
}
```

Но более гибким получится алгоритм, ищущий непосредственно последовательность символов, сравнивая их по одному, это позволяет искать не только словосочетания, но и, например, диалоги персонажей в книге. Данная ниже реализация может найти представленную в комментарии комбинацию, когда неверный символ слова является началом верной последовательности.

```java
private static boolean isInFile(String fileName, String search) throws
    IOException {
   FileInputStream fis = new FileInputStream(fileName);
   byte[] searchSequence = search.getBytes();
   int ch;
   int i = 0;         // geekgeekbrains
   while ((ch = fis.read()) != -1) {
      if (ch == searchSequence[i])
         i++;
      else {
         i = 0;
         if (ch == searchSequence[i]) i++;
      }
      if (i == searchSequence.length) {
         fis.close();
         return true;
```

```
16          }
17      }
18      fis.close();
19      return false;
20  }
```

Ещё более точной будет реализация, в которой формируется некоторое окно поиска, фактически, буфер, смещающийся посимвольно по читаемому файлу, это позволяет найти слово «гоголь» в комбинации «гогоголь».

```
1  private static boolean isInFile(String fileName, String search) throws
      IOException {
2     try (FileInputStream fis = new FileInputStream(fileName)) {
3        int ch;
4        StringBuilder sb = new StringBuilder();
5        while ((ch = fis.read()) != -1 && sb.length() < search.length())
            {
6           sb.append((char) ch);
7        }
8
9        do {
10          if (sb.toString().equals(search))
11             return true;
12          sb.deleteCharAt(0);
13          sb.append((char) ch);
14       } while ((ch = fis.read()) != -1);
15    } catch (IOException e) {
16          throw new RuntimeException(e);
17    }
18    return false;
19 }
```

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
  - Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
  - Посоветовать литературу на тему заданного вопроса;
  - Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Сохранение состояния приложения между запусками
- **Тайминг** 10-15 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Создать массив из 9 цифр и записать его в файл, используя поток вывода.

Вариант решения

```
int[] ar0 = {1,2,3,4,5,6,7,8,0,8,7,6,5,4,3};
final int DIGIT_BOUND = 48;

FileOutputStream fos = new FileOutputStream("save.out");
fos.write('[');
for (int i = 0; i < ar0.length; i++) {
   fos.write(DIGIT_BOUND + ar0[i]);
   if (i < ar0.length - 1) fos.write(',');
}
fos.write(']');
fos.flush();
fos.close();
```

$*_1$ Удостовериться, что числа записаны не символами, а цифрами, что сократит объём хранения в 8 и более раз (из-за представления цифр в виде ASCII символов). При этом важно помнить о допущениях такого способа записи, поскольку числа нужно как-то отделять друг от друга, а любой символ, например, пробел (32), имеет числовое представление, и внутри файла будет неотличим от числа 20. А любое отрицательное число будет воспринято потоком чтения как конец потока. Для выполнения задания сделать разделителем число 0.

Вариант решения

```
// assuming 0 is divider
int[] ar1 = {1,2,3,4,5,6,7,8,9,8,7,6,5,4,3};

FileOutputStream fos = new FileOutputStream("save0.out");
for (int i = 0; i < ar0.length; i++) {
   fos.write(ar1[i]);
   fos.write(0);
}
fos.flush();
fos.close();
```

Задание 2

- **Ценность этапа** Загрузка состояния приложения при запуске

- **Тайминг** 15-20 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Создать массив целых чисел и заполнить его информацией из файла, записанного в предыдущем задании.

    Вариант решения

```
int[] ar00 = new int[15];
final int DIGIT_BOUND = 48;

FileInputStream fis = new FileInputStream("save.out");
fis.read(); // '['
for (int i = 0; i < ar00.length; i++) {
    ar00[i] = fis.read() - DIGIT_BOUND;
    fis.read(); // ','
}
fis.close();

System.out.println(Arrays.toString(ar00));
```

  $*_1$ Прочитать данные из файла с числами, предполагая, что разделитель – это число 0.

    Вариант решения

```
int[] ar10 = new int[15];
// assuming 0 is divider

FileInputStream fis = new FileInputStream("save0.out");
int b;
int i = 0;
while ((b = fis.read()) != -1) {
    if (b != 0) {
        ar10[i++] = b;
    }
}
fis.close();

System.out.println(Arrays.toString(ar10));
```

Задание 3

- **Ценность этапа** Работа с текстом (автоматизированный поиск и замена текста в файле или группе файлов)
- **Тайминг** 10-15 мин
- **Действия преподавателя**
  - Выдать задание студентам;

- Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
- Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».

– **Задание**:
  - Написать программу заменяющую указанный символ в текстовом файле на пробел, сохраняющую получившийся текст в новый файл.

Вариант решения

```
FileInputStream fis = new FileInputStream("Main.java");
int i;
char what = ',';
char to = '!';
FileOutputStream fos = new FileOutputStream("Main.java.new");

while ((i = fis.read()) != -1) {
   if (i == what)
      fos.write(to);
   else
      fos.write(i);
}

fos.close();
```

$*_1$ Модифицировать алгоритм поиска замены символа так, чтобы программа осуществляла замену слова (последовательного набора символов) в исходном файле и записывала результат в новый файл.

Вариант решения

```
String search = "Hello";
String l = "Goodbye";
FileInputStream fis = new FileInputStream("Main.java");
FileOutputStream fos = new FileOutputStream("Main.java.new");

int ch;
StringBuilder sb = new StringBuilder();
while ((ch = fis.read()) != -1) {
   sb.append((char) ch);
   if (sb.length() == search.length()) {
      if (sb.toString().equals(search)) {
         fos.write(l.getBytes());
         sb.delete(0, search.length());
      } else {
         fos.write(sb.charAt(0));
         sb.deleteCharAt(0);
      }
   }
}
fos.write(sb.toString().getBytes());
```

Задание 4

- **Ценность этапа** Работа с файловой системой
- **Тайминг** 15-20 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Написать программу, читающую и выводящую в содержимое текущей папки .
    Вариант решения

```
int count = 0;
File path = new File( new File(".").getCanonicalPath() );
File[] dir = path.listFiles();
for (int i = 0; i < dir.length; i++) {
   if (dir[i].isDirectory()) continue;
   System.out.println(dir[i]);
}
```

$*_1$ Дописать программу таким образом, чтобы она рекурсивно выводила содержимое не только текущей папки, но и вложенных.
Вариант решения

```
void printContents(String path) throws IOException {
   int count = 0;
   File fullPath = new File(new File(path).getCanonicalPath() );
   File[] dir = fullPath.listFiles();
   for (int i = 0; i < dir.length; i++) {
      if (dir[i].isDirectory()) printContents(dir[i].toString());
      System.out.println(dir[i]);
   }
}
```

```
void printContents(".");
```

Задание 5 (необязательное)

- **Ценность этапа** Описание часто недостающих механик операционной системы по «массовой» работе с файлами.
- **Тайминг** 20-25 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».

— **Задание**: Написать функцию, добавляющую префикс к каждому из набора файлов, названия которых переданы ей в качестве параметров через пробел.
Вариант решения

```
String[] a = {"Main.java", "a.txt"};

for (String fileName : a) {
    Path file = Path.of(fileName);
    if (Files.exists(file)) {
        Files.move(file, Paths.get("pre_" + file), REPLACE_EXISTING);
    } else {
        System.out.printf("No file with name '%s'", fileName);
    }
}
```

Домашнее задание

— **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
— **Тайминг** 5-10 минут.
— **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки
  - Объяснить где взять проект заготовки для дз
— **Задания**

5-25 мин Выполнить все задания семинара, если они не были решены, без ограничений по времени;
Все варианты решения приведены в тексте семинара выше

15 мин 1. Написать функцию, создающую резервную копию всех файлов в директории (без поддиректорий) во вновь созданную папку `./backup`

```
Files.createDirectory(Path.of("./backup"));

DirectoryStream<Path> dir = Files.newDirectoryStream(Path.of("."));
for (Path file : dir) {
    if (Files.isDirectory(file)) continue;
    Files.copy(file, Path.of("./backup/" + file.toString()));
}
```

20-25 мин 2. Предположить, что числа в исходном массиве из 9 элементов имеют диапазон $[0, 3]$, и представляют собой, например, состояния ячеек поля для игры в крестикинолики, где 0 – это пустое поле, 1 – это поле с крестиком, 2 – это поле с ноликом, 3 – резервное значение. Такое предположение позволит хранить в одном числе типа `int` всё поле 3x3. Записать в файл 9 значений так, чтобы они заняли три байта.

```
int[] ar2 = {0,1,2,3,0,1,2,3,0};

FileOutputStream fos = new FileOutputStream("save1.out");
for (int b = 0; b < 3; b++) { // write to 3 bytes
    byte wr = 0;
```

```
    for (int v = 0; v < 3; v++) { // write by 3 values in each
        wr += (byte) (ar2[3 * b + v] << (v * 2));
    }
    fos.write(wr);
}
fos.flush();
fos.close();
```

20-25 мин 3. Прочитать числа из файла, полученного в результате выполнения задания 2.

```
int[] ar20 = new int[9];

FileInputStream fis = new FileInputStream("save1.out");
int b;
int i = 0;
while ((b = fis.read()) != -1) {
    for (int v = 0; v < 3; ++v) { // 3 values of four possible
        ar20[i++] = b >> (v * 2) & 0x3;
    }
}
fis.close();

System.out.println(Arrays.toString(ar20));
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

E. Семинар: Простейшие интерфейсы пользователя

E.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

E.2. Цели семинара

- Практика создания простых экранных форм – кнопки, компоненты форм;
- передача данных между формами и внутри формы;
- написание полноценного окна клиента чата;
- написание графической оболочки для настроек игры в крестики-нолики.

E.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучивает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	6-18	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (по минуте на ответ)
Рассмотрение ДЗ лекции	-5	19-23	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	24	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	15	25-30	Создание и компоновка элементов управления
Задание 2	15	31-35	Связь компонентов и сбор сведений о состоянии компонента
Задание 3	10	36-38	Передача данных с компонентов
Перерыв (если нужен)	5	39	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 4	15	40-44	Создание окна управления сервером
Задание 5	30	45-48	Создание окна клиентской части текстового чата
Домашнее задание	5	49-50	Объясните домашнее задание, подведите итоги урока

Что происходит	Время	Слайды	Описание
Рефлексия	10	51-52	Преподаватель запрашивает обратную связь
Длительность	120		

E.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5–7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
  - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
  - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Свойства окна, такие как размер и заголовок возможно задать (2)
     (a) написанием методов в классе-наследнике
     (b) вызовом методов в конструкторе
     (c) созданием констант в первых строках класса
  2. Для выполнения кода по нажатию кнопки на интерфейсе нужно (1)
     (a) создать обработчик кнопки и вписать код в него
     (b) переопределить метод нажатия у компонента кнопки
     (c) написать код непосредственно в методе создания кнопки
  3. Что такое Java Swing? (1)
     (a) набор библиотек для создания реактивных GUI;
     (b) среда разработки для Java;
     (c) язык программирования.
  4. Какая библиотека используется для создания графических интерфейсов на Java? (3)
     (a) Java Swing;
     (b) JavaFX;
     (c) обе библиотеки.

Рассмотрение ДЗ

- **Цель этапа:** Пояснить не очевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
  - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;

  - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
  - Полностью разобраться с кодом – это задание не нужно обсуждать, возможно просто спросить, кто действительно сел, взял в руки карандаш и блокнот, и попытался разобраться с кодом с лекции. Похвалить тех, кто это действительно сделал.
  - Переделать проверку победы, чтобы она не была реализована просто набором условий.

    Вариант решения

    На уроке был написан код, который, очевидно, никуда не годится

```
private boolean checkWin(int c) {
    if (field[0][0]==c && field[0][1]==c && field[0][2]==c) return true;
    if (field[1][0]==c && field[1][1]==c && field[1][2]==c) return true;
    if (field[2][0]==c && field[2][1]==c && field[2][2]==c) return true;

    if (field[0][0]==c && field[1][0]==c && field[2][0]==c) return true;
    if (field[0][1]==c && field[1][1]==c && field[2][1]==c) return true;
    if (field[0][2]==c && field[1][2]==c && field[2][2]==c) return true;

    if (field[0][0]==c && field[1][1]==c && field[2][2]==c) return true;
    if (field[0][2]==c && field[1][1]==c && field[2][0]==c) return true;
    return false;
}
```

    Задание подразумевало переписывание изменяющейся индексации в циклы, но, поскольку для игры в крестики-нолики нужно также иметь возможность работать на любом поле с любым числом фигур подряд для победы – имеет смысл рассматривать сразу следующее задание. Выполнивших это задание отдельно следует похвалить за точность выполнения задания.
  - Попробовать переписать логику проверки победы, чтобы она работала для поля 5х5 и количества фигур 4.

    Вариант решения

    более подробное объяснение алгоритма по ссылке

    Задание в первую очередь подразумевало необходимость составить алгоритм действий, прежде, чем писать какой-то код. Важно напомнить студентам о необходимости проектирования перед разработкой. Реализованный в качестве варианта решения алгоритм подразумевает проход по каждой клетке поля с проверкой на наличие линии по четырём направлениям (пошаговая проверка линии реализована методом `checkLine` с указанием координат начала проверки, направления и проверяемого символа) с проверкой на выход за пределы поля.

    Сначала учимся проверять одну линию в одном направлении. Если начинаем проверку от последнего поставленного символа – алгоритм сразу становится сложнее, потому что считать надо в 8 сторон, да ещё и учитывать, не был ли последний X в середине линии, но в итоге такой алгоритм на больших полях будет отрабатывать быстрее. Добавляем – из каких координат, какой символ, в каком направлении смотрим и проверки на выход за пределы поля (благо поле это глобальные переменные)

```
private boolean checkLine(int x, int y, int vx, int vy, int len, int
    c) {
    final int far_x = x + (len - 1) * vx;
```

```
    final int far_y = y + (len - 1) * vy;
    if (!isValidCell(far_x, far_y)) return false;
    for (int i = 0; i < len; i++) {
        if (field[y + i * vy][x + i * vx] != c) return false;
    }
    return true;
}
```

Затем осуществляем проверки линий вправо, вниз, вправо-вниз и вправо-вверх из каждой клетки поля, но с проверкой на выход за пределы.

```
private boolean checkWin(int c) {
    for (int i = 0; i < fieldSizeX; i++) {
        for (int j = 0; j < fieldSizeY; j++) {
            if (checkLine(i, j, 1, 0, winLength, c)) return true;
            if (checkLine(i, j, 1, 1, winLength, c)) return true;
            if (checkLine(i, j, 0, 1, winLength, c)) return true;
            if (checkLine(i, j, 1, -1, winLength, c)) return true;
        }
    }
    return false;
}
```

– ** Доработать искусственный интеллект, чтобы он мог примитивно блокировать ходы игрока, и примитивно пытаться выиграть сам.
Вариант решения
На лекции был представлен код, при исполнении которого, компьютер ходит в случайную клетку.

```
private void aiTurn() {
    int x, y;
    do {
        x = RANDOM.nextInt(fieldSizeX);
        y = RANDOM.nextInt(fieldSizeY);
    } while (!isEmptyCell(x, y));
    field[y][x] = DOT_AI;
}
```

Для того, чтобы компьютер мог проверить, может ли он победить и может ли он помешать человеку победить, нужно перед тем как делать ход случайным образом, добавить две проверки, которые должны завершиться выходом из метода хода компьютера, в случае успеха (то есть ход закончен, если мы успешно поставили выигрышную фигуру или успешно предотвратили один из вариантов победы игрока).

```
if (turnAIWinCell()) return;
if (turnHumanWinCell()) return;
```

Чтобы попытаться выиграть самому компьютер не будет делать ничего сложного – просто пройдёт по всему полю, попробует поставить последовательно в каждую клетку нолик и проверить, не выигрывает ли он. Если да – возвращает истину, если не выигрывает – стирает нолик и продолжает проверку.

```
private boolean turnAIWinCell() {
   for (int i = 0; i < fieldSizeY; i++) {
      for (int j = 0; j < fieldSizeX; j++) {
         if (isEmptyCell(j, i)) {
            field[i][j] = DOT_AI;
            if (checkWin(DOT_AI)) return true;
            field[i][j] = DOT_EMPTY;
         }
      }
   }
   return false;
}
```

Чтобы попытаться предотвратить победу человека – компьютер действует по тому же принципу – подставляет в каждую клетку крестик и если на поле победа человека – заменяет его на свой нолик и возвращает истину, считая, что успешно помешал человеку. Если же победы человека не происходит – продолжает проверку.

```
private boolean turnHumanWinCell() {
   for (int i = 0; i < fieldSizeY; i++) {
      for (int j = 0; j < fieldSizeX; j++) {
         if (isEmptyCell(j, i)) {
            field[i][j] = DOT_HUMAN;
            if (checkWin(DOT_HUMAN)) {
               field[i][j] = DOT_AI;
               return true;
            }
            field[i][j] = DOT_EMPTY;
         }
      }
   }
   return false;
}
```

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
  - Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
  - Посоветовать литературу на тему заданного вопроса;
  - Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом

материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Создание окна настроек игры в крестики-нолики, создание и компоновка элементов управления
- **Тайминг** 15-20 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - На лекции был написан фрейм, содержащий одну кнопку – начать игру и расположением самого окна настроек автоматически, относительно игрового окна.

```
public class SettingsWindow extends JFrame {
   private static final int WINDOW_HEIGHT = 230;
   private static final int WINDOW_WIDTH = 350;

   JButton btnStart = new JButton("Start new game");
   SettingsWindow(GameWindow gameWindow) {
      setLocationRelativeTo(gameWindow);
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
      btnStart.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            gameWindow.startNewGame(0, 3, 3, 3);
            setVisible(false);
         }
      });
      add(btnStart);
   }
}
```

Первое задание – добавить на экран компоновщик-сетку с одним столбцом и добавить над существующей кнопкой следующие компоненты в заданном порядке: `JLabel` с заголовком «Выберите режим игры», сгруппированные в `ButtonGroup` переключаемые `JRadioButton` с указанием режимов «Человек против компьютера» и «Человек против человека», `JLabel` с заголовком «Выберите размеры поля», `JLabel` с заголовком «Установленный размер поля:», `JSlider` со значениями 3..10, `JLabel` с заголовком «Выберите длину для победы», `JLabel` с заголовком «Установленная длина:», `JSlider` со значениями 3..10.

Вариант решения

Решение может быть выполнено сколь угодно гибко. Минимальное жизнеспособное решение приведено в листинге ниже.

```
setLayout(new GridLayout(10,1));
add(new JLabel("Выберите режимигры"));
ButtonGroup bg = new ButtonGroup();
```

```
4  JRadioButton pvc = new JRadioButton("Человек противкомпьютера");
5  JRadioButton pvp = new JRadioButton("Человек противчеловека");
6  bg.add(pvc);
7  bg.add(pvp);
8  add(pvc);
9  add(pvp);
10 add(new JLabel("Выберите размерыполя"));
11 add(new JLabel("Установленный размерполя:"));
12 add(new JSlider(3, 10, 3));
13 add(new JLabel("Выберите длинудляпобеды"));
14 add(new JLabel("Установленная длина:"));
15 add(new JSlider(3, 10, 3));
16 add(btnStart);
```

Если было получено такое решение, важно указать студентам, что мы не сможем снять показания с созданных таким образом компонентов и то, что будет использоваться, например, по кнопке, должно иметь более широкую область видимости, например, объявленным в классе

$*_1$ Сгруппировать объявление компонентов в два метода по смыслу – регулирование режима игры, регулирование параметров поля. Объявить компоненты так, чтобы они оказались доступны для обработчика кнопки.

Вариант решения

Код ниже. Общий смысл в том, чтобы получить доступные в классе компоненты и сделать конструктор не таким большим. Соответственно, методы должны быть вызваны из конструктора окна.

```
1  private JRadioButton humVSAI;
2  private JRadioButton humVShum;
3  private JSlider slideWinLen;
4  private JSlider slideFieldSize;
5
6  private void addGameModeControls() {
7     add(new JLabel("Выберите режимигры"));
8     humVSAI = new JRadioButton("Человек противкомпьютера");
9     humVShum = new JRadioButton("Человек противчеловека");
10    humVSAI.setSelected(true);
11    ButtonGroup gameMode = new ButtonGroup();
12    gameMode.add(humVSAI);
13    gameMode.add(humVShum);
14    add(humVSAI);
15    add(humVShum);
16 }
17
18 private void addFieldControls() {
19    JLabel lbFieldSize = new JLabel("Установленный размерполя:");
20    JLabel lbWinLength = new JLabel("Установленная длина:");
21    slideFieldSize = new JSlider(3, 10, 3);
22    slideWinLen = new JSlider(3, 10, 3);
23    add(new JLabel("Выберите размерыполя"));
24    add(lbFieldSize);
25    add(slideFieldSize);
```

```
    add(new JLabel("Выберите длинудляпобеды"));
    add(lbWinLength);
    add(slideWinLen);
}
```

$*_2$ Вынести неизменяемую часть изменяемых сообщений (подписи к слайдерам) в константы класса. Вынести размеры, применяемые в слайдерах в константы класса (избавиться от магических чисел). Вынести обработчик кнопки в отдельный метод.

Вариант решения

В результате должны появиться классовые константы, измениться начало метода `addFieldControls` и добавиться метод для обработчика кнопки, все изменения показаны в листинге

```
//constants
private static final int MIN_WIN_LENGTH = 3;
private static final int MIN_FIELD_SIZE = 3;
private static final int MAX_FIELD_SIZE = 10;
private static final String FIELD_SIZE_PREFIX = "Установленный
    размерполя: ";
private static final String WIN_LENGTH_PREFIX = "Установленная длина:
    ";

//addFieldControls
private void addFieldControls() {
    JLabel lbFieldSize = new JLabel(FIELD_SIZE_PREFIX);
    JLabel lbWinLength = new JLabel(WIN_LENGTH_PREFIX);
    slideFieldSize = new JSlider(MIN_FIELD_SIZE, MAX_FIELD_SIZE,
        MIN_FIELD_SIZE);
    slideWinLen = new JSlider(MIN_WIN_LENGTH, MAX_FIELD_SIZE,
        MIN_FIELD_SIZE);

//constructor
private final GameWindow gameWindow;
SettingsWindow(GameWindow gameWindow) {
    this.gameWindow = gameWindow;
    btnStart.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            btnStartDelegate();
        }
    });

//click delegate
private void btnStartDelegate() {
    gameWindow.startNewGame(0, 3, 3, 3);
    setVisible(false);
}
```

Задание 2

- **Ценность этапа** Создание окна настроек игры в крестики-нолики, связь компонентов и сбор сведений о состоянии компонента
- **Тайминг** 15-20 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Добавить компонентам интерактивности, а именно, при перемещении ползунка слайдера в соответствующих лейблах должны появляться текущие значения слайдеров. Для этого необходимо добавить к слайдеру слушателя изменений (как это было сделано для действия кнопки).

    Вариант решения

    Важно не забыть поменять начальные значения у лейблов, чтобы с видимостью экрана сразу становились видимы текущие настройки.

```java
JLabel lbFieldSize = new JLabel(FIELD_SIZE_PREFIX + MIN_FIELD_SIZE);
JLabel lbWinLength = new JLabel(WIN_LENGTH_PREFIX + MIN_FIELD_SIZE);
slideWinLen.addChangeListener(new ChangeListener() {
    @Override
    public void stateChanged(ChangeEvent e) {
        lbWinLength.setText(WIN_LENGTH_PREFIX + slideWinLen.getValue());
    }
});
slideFieldSize.addChangeListener(new ChangeListener() {
    @Override
    public void stateChanged(ChangeEvent e) {
        lbFieldSize.setText(FIELD_SIZE_PREFIX +
            slideFieldSize.getValue());
    }
});
```

$*_1$ Добавить автоматическое регулирование максимального значения у слайдера выигрышной длины при изменении значения слайдера размера поля.

```java
slideFieldSize.addChangeListener(new ChangeListener() {
    @Override
    public void stateChanged(ChangeEvent e) {
        int currentValue = slideFieldSize.getValue();
        lbFieldSize.setText(FIELD_SIZE_PREFIX + currentValue);
        slideWinLen.setMaximum(currentValue);
    }
});
```

Вариант решения

$*_2$ Добавить центрирование окна настроек относительно главного (родительского) окна. То есть, в центре родительского окна должен быть не левый верхний угол окна настроек (как это сделано сейчас), а также его центр.

Вариант решения
В решении важно увидеть отказ от использования метода, который использовался на лекции, потому что при помощи него решить задачу не удастся. Забираем у родительского окна его границы в виде прямоугольника, ищем центр этого прямоугольника по осям и отнимаем от полученных координат половину ширины и высоты окна настроек, соответственно, потому что метод `setLocation` будет устанавливать верхний-левый угол окна настроек, а не центр.

```
Settings(GameWindow mainWindow) {
    this.mainWindow = mainWindow;
    setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    //setLocationRelativeTo(mainWindow);
    Rectangle gameWindowBounds = mainWindow.getBounds();
    int posX = (int) gameWindowBounds.getCenterX() - WINDOW_WIDTH / 2;
    int posY = (int) gameWindowBounds.getCenterY() - WINDOW_HEIGHT / 2;
    setLocation(posX, posY);
    setResizable(false);
    //...
}
```

Задание 3

- **Ценность этапа** Передача данных с компонентов в окно игры
- **Тайминг** 10-15 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
- **Задание**:
  - В методе обработчика нажатия кнопки необходимо заменить константы в аргументе вызова метода старта игры на текущие показания компонентов (какая радио-кнопка активна, значение слайдера размеров поля, значение слайдера выигрышной длины).
    Вариант решения
    При решении этого задания важно понимать, что любое невозможное состояние компонента нельзя игнорировать и должно быть выведено сообщение об ошибке, идеально, если будет выброшено исключение. Необходимо пояснить студентам, что это делается для самого же разработчика, который спустя условные полгода не вспомнит, куда именно нужно добавлять код при разработке новой функциональности, например, нового режима игры, а исключение подскажет конкретную строку. Также в представленном ниже варианте решения используются константы, которые нужно добавить в класс Map.Почему именно в Map? потому что именно этот класс занимается логикой игры и если мы хотим добавить новый режим, например, игру в крестики-нолики втроём, то всю логику этого режима будет знать Map, а, следовательно, и константы с режимами должен задавать он.

```
public class Map extends JPanel {
    public static final int MODE_HVA = 0;
    public static final int MODE_HVH = 1;
    //...
 }
```

```
6
7  public class SettingsWindow extends JFrame {
8     //...
9  private void btnStartDelegate() {
10    int gameMode;
11    if (humVSAI.isSelected()) {
12       gameMode = Map.MODE_HVA;
13    } else if (humVShum.isSelected()) {
14       gameMode = Map.MODE_HVH;
15    } else {
16       throw new RuntimeException("Unknown game mode");
17    }
18    int fieldSize = slideFieldSize.getValue();
19    int winLength = slideWinLen.getValue();
20    gameWindow.startNewGame(gameMode, fieldSize, fieldSize, winLength);
21    setVisible(false);
22 }
```

Задание 4

- **Ценность этапа** Создание окна управления сервером
- **Тайминг** 10-15 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Создать простейшее окно управления сервером (по сути, любым), содержащее две кнопки (JButton) – запустить сервер и остановить сервер. Кнопки должны просто логировать нажатие (имитировать запуск и остановку сервера, соответственно) и выставлять внутри интерфейса соответствующее булево isServerWorking.
    Вариант решения
    В этом задании нет никаких подводных камней, просто экран, две кнопки, два обработчика

```
1  public class ServerWindow extends JFrame {
2     private static final int POS_X = 500;
3     private static final int POS_Y = 550;
4     private static final int WIDTH = 200;
5     private static final int HEIGHT = 100;
6
7     private final JButton btnStart = new JButton("Start");
8     private final JButton btnStop = new JButton("Stop");
9     private boolean isServerWorking;
10
11    public static void main(String[] args) {
12       new ServerWindow();
13    }
```

```
    private ServerWindow() {
        isServerWorking = false;
        btnStop.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                isServerWorking = false;
                System.out.println("Server stopped " + isServerWorking);
            }
        });

        btnStart.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                isServerWorking = true;
                System.out.println("Server started " + isServerWorking);
            }
        });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setBounds(POS_X, POS_Y, WIDTH, HEIGHT);
        setResizable(false);
        setTitle("Chat server");
        setAlwaysOnTop(true);
        setLayout(new GridLayout(1, 2));
        add(btnStart);
        add(btnStop);
        setVisible(true);
    }
}
```

$*_1$ Если сервер не запущен, кнопка остановки должна сообщить, что сервер не запущен и более ничего не делать. Если сервер запущен, кнопка старта должна сообщить, что сервер работает и более ничего не делать.

Вариант решения

Это задание также без подводных камней, важно не забыть, что после выдачи сообщения в проверке нужно прекратить выполнение обработчика.

```
btnStop.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (!isServerWorking) {
            System.out.println("already stopped");
            return;
        }
        isServerWorking = false;
        System.out.println("Server stopped " + isServerWorking);
    }
});

btnStart.addActionListener(new ActionListener() {
```

```
14        @Override
15        public void actionPerformed(ActionEvent e) {
16            if (isServerWorking) {
17                System.out.println("already working");
18                return;
19            }
20            isServerWorking = true;
21            System.out.println("Server started " + isServerWorking);
22        }
23    });
```

*₂ Добавить на окно компонент JtextArea и выводить сообщения сервера в него, а не в терминал.

Вариант решения

В данном задании важно, что при добавлении текстовой области изменяется лейаут окна (обратно на стандартный), а кнопки довольно логично смотрятся сверху или снизу на отдельной панели с компоновкой сеткой. Также все выводы в консоль заменяются на вызов метода log.append(), и это поведение желательно убрать в приватный метод, что-то вроде putLog(String msg), потому что сейчас мы хотим логировать так, а дальше захотим иначе и не надо будет искать во всём коде – что и как логируется.

```
1  private static final int WIDTH = 400;
2  private static final int HEIGHT = 300;
3
4  private final JTextArea log = new JTextArea();
5
6  private ServerWindow() {
7      isServerWorking = false;
8      btnStop.addActionListener(new ActionListener() {
9          @Override
10         public void actionPerformed(ActionEvent e) {
11             if (!isServerWorking) {
12                 log.append("already stopped\n");
13                 return;
14             }
15             isServerWorking = false;
16             log.append("Server stopped " + isServerWorking + "\n");
17         }
18     });
19
20     btnStart.addActionListener(new ActionListener() {
21         @Override
22         public void actionPerformed(ActionEvent e) {
23             if (isServerWorking) {
24                 log.append("already working\n");
25                 return;
26             }
27
28             isServerWorking = true;
29             log.append("Server started " + isServerWorking + "\n");
```

```
30        }
31     });
32 
33     setDefaultCloseOperation(EXIT_ON_CLOSE);
34     setBounds(POS_X, POS_Y, WIDTH, HEIGHT);
35     setResizable(false);
36     setTitle("Chat server");
37     setAlwaysOnTop(true);
38 
39     JPanel panelTop = new JPanel(new GridLayout(1, 2));
40     log.setEditable(false);
41     log.setLineWrap(true);
42     JScrollPane scrollLog = new JScrollPane(log);
43     panelTop.add(btnStart);
44     panelTop.add(btnStop);
45     add(panelTop, BorderLayout.NORTH);
46     add(scrollLog, BorderLayout.CENTER);
47 
48     setVisible(true);
49 }
```

Задание 5

- **Ценность этапа** Создание окна клиентской части текстового чата
- **Тайминг** 20-25 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Создать окно клиента чата. Окно должно содержать `JtextField` для ввода логина, пароля, IP-адреса сервера, порта подключения к серверу, область ввода сообщений, `JTextArea` область просмотра сообщений чата и `JButton` подключения к серверу и отправки сообщения в чат. Желательно сразу сгруппировать компоненты, относящиеся к серверу сгруппировать на `JPanel` сверху экрана, а компоненты, относящиеся к отправке сообщения – на `JPanel` снизу.

    Вариант решения

    К решению, как и к проверке желательно подойти последовательно, сначала создать пустое окно с размерами и со стандартными элементами вроде заголовка и обработки закрытия окна на крестик

```
1 public class ClientGUI extends JFrame {
2     private static final int WIDTH = 400;
3     private static final int HEIGHT = 300;
4 
5     ClientGUI() {
6         setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
7         setLocationRelativeTo(null);
```

```
        setSize(WIDTH, HEIGHT);
        setTitle("Chat Client");

        setVisible(true);
    }

    public static void main(String[] args) {
        new ClientGUI();
    }

}
```

Далее добавить компоненты, касающиеся сервера и панель, например, верхнюю. В конструкторе, добавить эти элементы на эту панель и саму панель на экран.

```
private final JPanel panelTop = new JPanel(new GridLayout(2, 3));
private final JTextField tfIPAddress = new JTextField("127.0.0.1");
private final JTextField tfPort = new JTextField("8189");
private final JTextField tfLogin = new JTextField("ivan_igorevich");
private final JPasswordField tfPassword = new JPasswordField("123456");
private final JButton btnLogin = new JButton("Login");
// constructor
     panelTop.add(tfIPAddress);
     panelTop.add(tfPort);
     panelTop.add(tfLogin);
     panelTop.add(tfPassword);
     panelTop.add(btnLogin);
     add(panelTop, BorderLayout.NORTH);
```

Дальше делаем нижнюю панельку и также добавляем её на экран.

```
private final JPanel panelBottom = new JPanel(new BorderLayout());
private final JTextField tfMessage = new JTextField();
private final JButton btnSend = new JButton("Send");
// constructor
     panelBottom.add(tfMessage, BorderLayout.CENTER);
     panelBottom.add(btnSend, BorderLayout.EAST);
     add(panelBottom, BorderLayout.SOUTH);
```

И как без внимания оставить лог. Его нужно сделать не редактируемым и прокручиваемым. для этого вызовем метод `setEditable` и добавим в `ScrollBox`.

```
private final JTextArea log = new JTextArea();
// constructor
      log.setEditable(false);
      JScrollPane scrollLog = new JScrollPane(log);
      add(scrollLog);
```

$*_1$ Добавить на экран компонент `JList<String>` – имитацию списка пользователей, заполнить этот список несколькими выдуманными именами пользователей чата. Подсказка: компонент не может добавлять или убирать элементы списка, он работает с методом `setListData()`, изучите его аргументы.

Вариант решения

В самом создании списка нет ничего необычного, кроме использования метода из подсказки. Также достаточно важно, что список необходимо поместить в область прокрутки, поскольку список пользователей чата может не поместиться на экран.

```
private final JList<String> userList = new JList<>();
// constructor
      JScrollPane scrollUsers = new JScrollPane(userList);
      add(scrollUsers, BorderLayout.EAST);
      String users[] = {"user1", "user2", "user3", "user4", "user5",
         "user6", "user7", "user8", "user9", "user10"};
      userList.setListData(users);
```

если какой-то из пользователей внезапно выберет достаточно длинное имя, оно может перекрыть область вывода сообщений, это обусловлено приоритетом компоновки (центр имеет наименьший приоритет). Для того, чтобы этого избежать нужно для области прокрутки выставить предпочтительный размер (например, ширина 100 и высота не важна).

```
      scrollUsers.setPreferredSize(new Dimension(100, 0));
```

Домашнее задание

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.
- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки
  - Объяснить где взять проект заготовки для дз
- **Задания**

5-25 мин Выполнить все задания семинара, если они не были решены, без ограничений по времени;

Все варианты решения приведены в тексте семинара выше

15 мин 1. Отправлять сообщения из текстового поля сообщения в лог по нажатию кнопки или по нажатию клавиши Enter на поле ввода сообщения.

```
// constructor
btnSend.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      sendMsg();
   }
});
tfMessage.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      sendMsg();
   }
});
```

```
14
15    private void sendMsg() {
16       String msg = tfMessage.getText(); // получилисообщение
17       String username = tfLogin.getText(); // получилиимя
18       if ("".equals(msg)) return; // сравнилинапустое
19       log.append(username + ": " + msg + "\n"); // добавили
20       tfMessage.setText(null); // очищаем
21       tfMessage.requestFocusInWindow(); // сфокусируемся
22    }
```

10-15 мин 2. Продублировать импровизированный лог (историю) чата в файле.

```
1    private void sendMsg() {
2       //...
3       try (FileWriter out = new FileWriter("log.txt", true)) {
4          out.write(username + ": " + msg + "\n");
5          out.flush();
6       } catch (IOException e) {
7          throw new RuntimeException(e);
8       }
9    }
```

15-20 мин 3. При запуске клиента чата заполнять поле истории из файла, если он существует. Обратите внимание, что чаще всего история сообщений хранится на сервере и заполнение истории чата лучше делать при соединении с сервером, а не при открытии окна клиента.

```
1  // constructor
2  try (FileInputStream fis = new FileInputStream("log.txt")) {
3     int b;
4     while ((b = fis.read()) != -1) {
5        log.append(Character.valueOf((char) b).toString());
6     }
7  } catch (IOException e) {
8     e.printStackTrace();
9  }
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

Ж. Семинар: Интерфейсы и API

Ж.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

Ж.2. Цели семинара

- Практика создания и проектирования интерфейсов;
- передача и обработка сообщений с использованием интерфейсов;
- Применение существующих интерфейсов для обработки исключений;
- Отделение графического, сетевого и логического слоёв для приложения «Сетевой чат».

Ж.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучивает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	6-18	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (по минуте на ответ)
Рассмотрение ДЗ лекции	10	19-24	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	25	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	30	26-30	Отделение бизнес-логики и графического интерфейса
Перерыв (если нужен)	5	31	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 2	30	32-36	Создание и частичная реализация интерфейсов для улучшения понимания механизмов взаимодействия объектов
Задание 3	10	37-40	Написать абстрактный пример применения интерфейсов с реализацией по умолчанию и наследование
Домашнее задание	5	41-42	Объясните домашнее задание, подведите итоги урока

Что происходит	Время	Слайды	Описание
Рефлексия	10	43-44	Преподаватель запрашивает обратную связь
Длительность	120		

Ж.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
    - Запрашивает активность от аудитории в чате;
    - Презентует цели курса и семинара;
    - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5–7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
    - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
    - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
    1. Множественное наследование в Java (3)
        (a) запрещено;
        (b) разрешено;
        (c) разрешено для интерфейсов.
    2. Интерфейсы позволяют (2)
        (a) удобно создавать новые объекты, не связанные наследованием;
        (b) единообразно обращаться к методам объектов, не связанных наследованием;
        (c) полностью заменить наследование.
    3. Поле в интерфейсе (2)
        (a) невозможно;
        (b) public static final;
        (c) private final.
    4. Обработчик исключений для графического потока (2)
        (a) это статический метод;
        (b) это интерфейс;
        (c) реализован JVM.

Рассмотрение ДЗ

- **Цель этапа:** Пояснить не очевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
    - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;

  - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
  - Полностью разобраться с кодом – это задание не нужно обсуждать, возможно просто спросить, кто действительно сел, взял в руки карандаш и блокнот, и попытался разобраться с кодом с лекции. Похвалить тех, кто это действительно сделал.
  - Для приложения с шариками описать появление и убирание шариков по клику мышки левой и правой кнопкой соответственно

    Вариант решения

    На лекции был описан инициализатор приложения, заполняющий массив игровых объектов шариками. необходимо было описать слушатель мышки, добавляющий и убирающий шарики из массива

```
if (e.getButton() == MouseEvent.BUTTON1) {
    interactables[objectsCount++] = new Ball(e.getX(), e.getY());
} else if (e.getButton() == MouseEvent.BUTTON3) {
    if (objectsCount == 1) return;
    objectsCount--;
}
```

Для обеспечения работоспособности данного кода необходимо добавить в мячик конструктор, принимающий начальные координаты

```
Ball(int x, int y) {
    this();
    this.x = x;
    this.y = y;
}
```

  - Написать, выбросить и обработать такое исключение, которое не позволит создавать более, чем 15 шариков.

    Вариант решения

    Очевидно, что обработка исключений должна вызвать ряд проблем, поскольку тема обработки исключений при использовании графических интерфейсов рассматривается только на следующей лекции, но за попытку решить проблему следует похвалить студентов.

Листинг 158: Собственно, исключение

```
public class BallsOverflowException extends RuntimeException {
    BallsOverflowException() {
        super("Balls overflow, 15 allowed!");
    }
}
```

По очевидным причинам выбрасывать исключение там, где добавляются шарики – не логично, поэтому необходимо выбрасывать или на апдейте или на рендере.

Листинг 159: Место выброса исключения

```
private void update(MainCanvas canvas, float deltaTime) {
    for (int i = 0; i < objectsCount; i++) {
        interactables[i].update(canvas, deltaTime);
```

```
    }
    if (objectsCount >= 15)
        throw new BallsOverflowException();
}
```

Для обработки исключений в Swing необходимо установить обработчик исключений для потока. сделаем этим обработчиком «себя».

Листинг 160: Обработка исключения

```
private MainWindow() { //constructor
    Thread.setDefaultUncaughtExceptionHandler(this);
}

@Override
public void uncaughtException(Thread t, Throwable e) {
    if (e.getClass().equals(BallsOverflowException.class)) {
        System.out.println(e.getMessage());
    }
}
```

– ** Написать ещё одно приложение, в котором на белом фоне будут перемещаться изображения формата png, лежащие в папке проекта.
Вариант решения
Главным отличием приложения будет работа с изображением. Такова была идея примера – тиражируемость приложения. То есть вообще весь код может быть взят из шариков или квадратиков, но в главном рисуемом объекте должен происходить не рендеринг примитива, а рендеринг картинки.

```

public class Ball extends Sprite {
    private Image img = null;
    private float vX;
    private float vY;

    Ball() {
        halfHeight = 64; // logo size magic numbers
        halfWidth = 62;
        try {
            img = ImageIO.read(new File("logo.png"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        vX = 100f + (float) (Math.random() * 200f);
        vY = 100f + (float) (Math.random() * 200f);
    }

    @Override
    public void render(MainCanvas canvas, Graphics g) {
        g.drawImage(img, (int) getLeft(), (int) getTop(), null);
    }
}
```

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
  - Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
  - Посоветовать литературу на тему заданного вопроса;
  - Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Отделение бизнес-логики и графического интерфейса.
- **Тайминг** 25-30 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Пояснить студентам пользу и необходимость следования паттерну MVC, проговорить важность разделения логики и открывающиеся возможности при правильном проектировании.
- **Задание**:
  - На предыдущем семинаре было описано окно сервера приложения, содержащее две кнопки (старт и стоп) и текстовое поле журнала. Необходимо вынести логику работы сервера в класс `ChatServer`, а в обработчиках кнопок оставить только логику нажатия кнопки и журналирования сообщений от сервера.
    Для достижения цели необходимо описать интерфейс «слушатель сервера», с методом «получить сообщение», вызывать его с одной стороны, и реализовать с другой.
    Вариант решения

```
//ChatServerListener.java
package ru.gb.jdk.two.sem;

public interface ChatServerListener {
    void onMessageReceived(String msg);
}

//ChatServer.java

package ru.gb.jdk.two.sem;

public class ChatServer {
    private boolean isServerWorking;
```

```
14      private final ChatServerListener listener;
15
16      ChatServer(ChatServerListener listener) {
17          isServerWorking = false;
18          this.listener = listener;
19      }
20
21      public void start() {
22          if (isServerWorking) {
23              listener.onMessageReceived("Server is already working");
24              return;
25          }
26          listener.onMessageReceived("Server started");
27          isServerWorking = true;
28      }
29
30      public void stop() {
31          if (!isServerWorking) {
32              listener.onMessageReceived("Server is stopped");
33              return;
34          }
35          listener.onMessageReceived("Server stopped");
36          isServerWorking = false;
37      }
38  }
39
40  //ServerWindow.java
41  //constructor
42  server = new ChatServer(this);
43
44  btnStop.addActionListener(new ActionListener() {
45      @Override
46      public void actionPerformed(ActionEvent e) {
47          server.stop();
48      }
49  });
50
51  btnStart.addActionListener(new ActionListener() {
52      @Override
53      public void actionPerformed(ActionEvent e) {
54          server.start();
55      }
56    });
57
58  // method implementation
59  @Override
60  public void onMessageReceived(String msg) {
61      log.append(msg + "\n");
62  }
```

$*_1$ Отделить функционал логгирования сообщений сервера для простоты изменения

способа оповещения пользователя.

Вариант решения

```
@Override
public void onMessageReceived(String msg) {
    putMessageToLog(msg);
}

private void putMessageToLog(String msg) {
    log.append(msg + "\n");
}
```

Задание 2

- **Ценность этапа** Создание и частичная реализация интерфейсов для улучшения понимания механизмов взаимодействия объектов.
- **Тайминг** 25-30 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Создать интерфейсы `ServerSocketThreadListener` и `SocketThreadListener`, содержащие методы, соответствующие событиям сервера и клиента чата. Реализовать созданные интерфейсы простым логированием. Со стороны клиента – только `SocketThreadListener`, со стороны сервера – оба интерфейса.

    Вариант решения

```
package ru.gb.jdk.two.sem.network;

import java.net.Socket;

public interface SocketThreadListener {
    void onSocketStart(Socket s);
    void onSocketStop();

    void onSocketReady(Socket socket);
    void onReceiveString(Socket s, String msg);

    void onSocketException(Throwable e);
}

package ru.gb.jdk.two.sem.network;

import java.net.ServerSocket;
import java.net.Socket;

```

```
public interface ServerSocketThreadListener {
    void onServerStart();
    void onServerStop();
    void onServerSocketCreated(ServerSocket s);
    void onServerSoTimeout(ServerSocket s);
    void onSocketAccepted(ServerSocket s, Socket client);
    void onServerException(Throwable e);
}
```

```
 // Server

/**
 * Server socket thread methods
 * */
@Override
public void onServerStart() { listener.onMessageReceived("Server
    thread started"); }

@Override
public void onServerStop() { listener.onMessageReceived("Server thread
    stopped"); }

@Override
public void onServerSocketCreated(ServerSocket s) {
    listener.onMessageReceived("Server socket created"); }

@Override
public void onServerSoTimeout(ServerSocket s) {
    // listener.onMessageReceived("Accept timeout");
}

@Override
public void onSocketAccepted(ServerSocket s, Socket client) {
    listener.onMessageReceived("client connected"); }

@Override
public void onServerException(Throwable e) { e.printStackTrace(); }

/**
 * Socket Thread listening
 * */
@Override
public synchronized void onSocketStart(Socket s) {
    listener.onMessageReceived("Client connected"); }

@Override
public synchronized void onSocketStop() {
    listener.onMessageReceived("Client dropped"); }

@Override
```

```
36 public synchronized void onSocketReady(Socket socket) {
       listener.onMessageReceived("Client is ready"); }
37
38 @Override
39 public synchronized void onReceiveString(Socket s, String msg) {
       listener.onMessageReceived(msg); }
40
41 @Override
42 public void onSocketException(Throwable e) { e.printStackTrace(); }
43
44 // Client
45 @Override
46 public void onSocketStart(Socket s) { log.append("Started" + "\n"); }
47
48 @Override
49 public void onSocketStop() { log.append("Stopped" + "\n"); }
50
51 @Override
52 public void onSocketReady(Socket socket) { log.append("Ready" + "\n");
       }
53
54 @Override
55 public void onReceiveString(Socket s, String msg) { log.append(msg +
       "\n"); }
56
57 @Override
58 public void onSocketException(Throwable e) { e.printStackTrace(); }
```

$*_1$ Создать классы – ServerSocketThread и SocketThread, соответственно слушателям, то есть реализовать в классе конструкторы с возможностью сохранения ссылки на слушателей. Создать экземпляр ServerSocketThread из объекта ChatServer, а SocketThread из ClientGUI. На данном этапе, не важно, где именно будут создаваться эти объекты.

Вариант решения

```
1  package ru.gb.jdk.two.sem.network;
2
3  public class ServerSocketThread {
4     private ServerSocketThreadListener listener;
5
6     public ServerSocketThread(ServerSocketThreadListener listener) {
7        this.listener = listener;
8     }
9  }
10
11 package ru.gb.jdk.two.sem.network;
12
13 public class SocketThread {
14    private final SocketThreadListener listener;
15
16    public SocketThread(SocketThreadListener listener) {
```

```
        this.listener = listener;
    }
}

// Server
private ServerSocketThread server;


public void start() {
    if (isServerWorking) {
        listener.onMessageReceived("Server is already working");
        return;
    }
    server = new ServerSocketThread(this);
    listener.onMessageReceived("Server started");
    isServerWorking = true;
}

//Client
private void connect() {
    SocketThread socketThread = new SocketThread(this);
}
```

Задание 3

- **Ценность этапа** Написать абстрактный пример применения интерфейсов с реализацией по умолчанию и наследованием.
- **Тайминг** 10-15 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
- **Задание**:
  - Описать команду разработчиков. В команде разработчиков могут находиться бэкендеры, которые в состоянии писать серверный код, фронтендеры, которые могут программировать экранные формы, и фуллстэк разработчики, совмещающие в себе обе компетенции. Реализовать класс фулстэк разработчика, создать экземпляр и последовательно вызвать все его методы.

    Вариант решения

```
package ru.gb.jdk.two.sem.devs;

public interface Backender { void developServer(); }

public interface Frontender { void developGUI(); }

public interface Fullstack extends Backender, Frontender {
}

public class FullstackDeveloper implements Fullstack {
```

```
11      @Override
12      public void developServer() { System.out.println("Server done"); }
13
14      @Override
15      public void developGUI() { System.out.println("GUI done"); }
16  }
17
18  public class Main {
19      public static void main(String[] args) {
20          FullstackDeveloper dev = new FullstackDeveloper();
21          dev.developGUI();
22          dev.developServer();
23      }
24  }
```

$*_1$ Добавить возможность при создании экземпляров классов любых интерфейсов – добавлять их в один массив, например, `Developer[] team = {...};`
Вариант решения
К первому решению добавляется маркерный интерфейс «Разработчик». Очевидно, что нельзя для этого интерфейса вызвать более точные методы фуллстэк разработчика.

```
1   package ru.gb.jdk.two.sem.devs;
2
3   public interface Developer {
4   }
5
6   public interface Backender extends Developer { void developServer(); }
7
8   public interface Frontender extends Developer { void developGUI(); }
9
10  public class Main {
11      public static void main(String[] args) {
12          Developer dev = new FullstackDeveloper();
13      }
14  }
```

Домашнее задание

- **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями. В данном семинаре домашнее задание не предусмотрено в связи со сложностью заданий семинара. Необходимо досконально разобраться в написанном (и запланированном коде) и взаимосвязях объектов. Также возможно уточнить, что задания семинара направлены на написание многопоточного сетевого чата, но, поскольку это будет достаточно сложный проект, его написание приводится постепенно
- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки

- Объяснить где взять проект заготовки для дз

- **Задания**

5-25 мин Выполнить все задания семинара, если они не были решены, без ограничений по времени;

Все варианты решения приведены в тексте семинара выше

5 мин 1. Дописать третье задание таким образом, чтобы в идентификатор типа `Developer` записывался объект `Frontender`, и далее вызывался метод `developGUI()`, не изменяя существующие интерфейсы, только код основного класса.

```
public class Main {
   public static void main(String[] args) {
      Developer dev = new FrontendDeveloper();
      if (dev instanceof Frontender) {
         ((Frontender) dev).developGUI();
      }
   }
}
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

3. Семинар: Обобщения

3.1. Инструментарий

- Презентация для преподавателя, ведущего семинар;
- Фон GeekBrains для проведения семинара в Zoom;
- JDK любая 11 версии и выше;
- IntelliJ IDEA Community Edition для практики и примеров используется IDEA.

3.2. Цели семинара

- Практика создания простых обобщений;
- Создание собственного обобщённого контейнера с описанием базовых алгоритмов (введение в коллекции);
- Изучение поведения объекта итератора;
- Закрепление понимания механизмов работы коллекций.

3.3. План-содержание

Что происходит	Время	Слайды	Описание
Организационный момент	5	1-5	Преподаватель ожидает студентов, поддерживает активность и коммуникацию в чате, озвучивает цели и планы на семинар. Важно упомянуть, что выполнение домашних заданий с лекции является, фактически, подготовкой к семинару
Quiz	5	6-18	Преподаватель задаёт вопросы викторины, через 30 секунд демонстрирует слайд-подсказку и ожидает ответов (по минуте на ответ)
Рассмотрение ДЗ лекции	10	19-23	Преподаватель демонстрирует свой вариант решения домашнего задания с лекции, возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов
Вопросы и ответы	10	24	Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы
Задание 1	10	25-27	Формирование навыков базовой работы с обобщениями
Задание 2	20	28-30	Практика написания обобщённого кода с дополнительным описанием алгоритмов манипулирования данными
Перерыв (если нужен)	5	31	Преподаватель предлагает студентам перерыв на 5 минут (студенты голосуют)
Задание 3	20	32-34	Изучение поведения итератора для понимания механизмов его действия и необходимости его использования в работе

Что происходит	Время	Слайды	Описание
Задание 4	20	35-38	Закрепление понимания механизмов работы коллекций (в том числе с применением итераторов)
Домашнее задание	5	39-40	Объясните домашнее задание, подведите итоги урока
Рефлексия	10	41-42	Преподаватель запрашивает обратную связь
Длительность	120		

3.4. Подробности

Организационный момент

- **Цель этапа:** Позитивно начать урок, создать комфортную среду для обучения.
- **Тайминг:** 3-5 минут.
- **Действия преподавателя:**
  - Запрашивает активность от аудитории в чате;
  - Презентует цели курса и семинара;
  - Презентует краткий план семинара и что студент научится делать.

Quiz

- **Цель этапа:** Вовлечение аудитории в обратную связь.
- **Тайминг:** 5–7 минут (4 вопроса, по минуте на ответ).
- **Действия преподавателя:**
  - Преподаватель задаёт вопросы викторины, представленные на слайдах презентации;
  - через 30 секунд демонстрирует слайд-подсказку и ожидает ответов.
- **Вопросы и ответы:**
  1. Какая основная цель использования Java generics? (1)
     (a) Упрощение программирования и повышение безопасности типов
     (b) Ускорение работы программы и сокращение объема кода
     (c) Позволяют работать с различными базами данных одновременно
  2. Что такое параметризованный класс в Java generics? (1)
     (a) Класс, который принимает параметр типа
     (b) Класс, который имеет только один параметр типа
     (c) Класс, который можно использовать только с определенным типом данных
  3. Какие ограничения можно использовать с wildcard в Java generics? (1)
     (a) extends и super
     (b) only
     (c) extends
  4. Можно ли создать экземпляр обобщенного типа в Java, внутри класса, описывающего этот тип? (2)
     (a) Да, это возможно
     (b) Нет, такое создание экземпляров обобщенных типов запрещено
     (c) Это зависит от контекста использования обобщенного типа

Рассмотрение ДЗ

- **Цель этапа:** Пояснить не очевидные моменты в формулировке ДЗ с лекции, синхронизировать прочитанный на лекции материал к началу семинара.
- **Тайминг:** 15-20 минут.
- **Действия преподавателя:**
  - Преподаватель демонстрирует свой вариант решения домашнего задания из лекции;
  - возможно, по предварительному опросу, демонстрирует и разбирает вариант решения одного из студентов.
- **Домашнее задание из лекции:**
  - Написать метод, который меняет два элемента массива местами (массив может быть любого ссылочного типа);

    Вариант решения

    Задание с подвохом, при решении необязательно было использовать дженерики, формулировка задания не ограничивает используемые в массиве типы, наоборот, явно проговаривается, что массив может быть любого типа.

```java
private static void swap(Object[] arr, int from, int to) {
   Object temp = arr[from];
   arr[from] = arr[to];
   arr[to] = temp;
}

public static void main(String[] args) {
   Object[] arr = {1, 2.0f, "hello"};
   System.out.println(Arrays.toString(arr));
   swap(arr, 0, 2);
   System.out.println(Arrays.toString(arr));
}
```

  - Большая задача:
    * Есть классы `Fruit` -> `Apple`, `Orange`; (больше не надо);
    * Класс `Box` в который можно складывать фрукты, коробки условно сортируются по типу фрукта, поэтому в одну коробку нельзя сложить и яблоки, и апельсины; Для хранения фруктов внутри коробки можете использовать `ArrayList`;
    * Сделать метод `getWeight()` который высчитывает вес коробки, зная количество фруктов и вес одного фрукта(вес яблока – `1.0f`, апельсина – `1.5f`, не важно в каких единицах);
    * Внутри класса коробки сделать метод `compare()`, который позволяет сравнить текущую коробку с той, которую подадут в `compare()` в качестве параметра, `true` – если их веса равны, `false` в противном случае (коробки с яблоками возможно сравнивать с коробками с апельсинами);
    * Написать метод, который позволяет пересыпать фрукты из текущей коробки в другую коробку (при этом, нельзя яблоки высыпать в коробку с апельсинами), соответственно, в текущей коробке фруктов не остается, а в другую перекидываются объекты, которые были в этой коробке.

    Вариант решения Создали фрукты с каким то разным весом, создали коробку, которая может вместить только фрукты. Надо фрукты там как то хранить, поэтому – список. Соответственно при создании можем в нашу коробку какие то фрукты накидать. Соответственно метод, который будет уметь добавлять фрукты в коробку

```java
private interface Fruit { float getWeight(); }
```

```
private static class Apple implements Fruit {
    static final float weight = 1.0f;
    @Override public float getWeight() { return weight; }
}
private static class Orange implements Fruit {
    static final float weight = 1.5f;
    @Override public float getWeight() { return weight; }
}
public static class Box<T extends Fruit> {
    private final List<T> container;
    Box(T[] init) {
        container = new ArrayList<>();
        for (T f : init) { add(f); }
    }
    void add(T fruit) { container.add(fruit); }
    void print() { System.out.println(getWeight()); }
}
```

Поскольку в коробке у нас могут быть только одно типа фрукты – мы можем не перебирать каждый из них, а умножить вес одного на количество.

```
float getWeight() {
    return (container.isEmpty())
          ? 0
          : container.get(0).getWeight() * container.size();
}
```

Можно было, по большому счёту, переопределить метод equals() или написать свой метод compare() который принимает вторую коробку и сравнивает вес. Сравнивать можно любые фрукты, значит в методе не подходит параметр типа <T>

```
boolean compare(Box<?> with) {
    return this.getWeight() - with.getWeight() < 0.0001;
}
```

Осталось только уметь пересыпать из одной коробки в другую не смешивая. На входе должна быть коробка того же типа, что и у объекта, и тут два пути, либо пересыпать всё из текущей коробки в другую, либо из другой в текущую, метод назван `transferTo()`, поэтому пересыпаться фрукты будут из текущей коробки. Возникает вопрос: если создаётся коробка с яблоками `Box<Apple>` и коробка с фруктами `Box<Fruit>`, должна ли быть возможность перекинуть из яблочной во фруктовую? Однозначно, нет. Поэтому, в аргументе метода используется `Box<? super T>`, таким образом яблоки возможно кидать в яблоки или в яблочных родителей, но не в апельсины.

```
void transferTo(Box<? super T> dest) {
    dest.container.addAll(container);
    container.clear();
}
```

Вопросы и ответы

- **Ценность этапа** Вовлечение аудитории в обратную связь, пояснение неочевидных моментов в материале лекции и другой проделанной работе.
- **Тайминг** 5-15 минут
- **Действия преподавателя**
  - Преподаватель ожидает вопросов по теме прошедшей лекции, викторины и продемонстрированной работы;
  - Если преподаватель затрудняется с ответом, необходимо мягко предложить студенту ответить на его вопрос на следующем семинаре (и не забыть найти ответ на вопрос студента!);
  - Предложить и показать пути самостоятельного поиска студентом ответа на заданный вопрос;
  - Посоветовать литературу на тему заданного вопроса;
  - Дополнительно указать на то, что все сведения для выполнения домашнего задания, прохождения викторины и работы на семинаре были рассмотрены в методическом материале к этому или предыдущим урокам.

Задание 1

- **Ценность этапа** Формирование навыков базовой работы с обобщениями.
- **Тайминг** 10-15 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Пояснить студентам пользу и необходимость следования паттерну MVC, проговорить важность разделения логики и открывающиеся возможности при правильном проектировании.
- **Задание**:
  - Создать обобщенный класс с тремя параметрами (T, V, K). Класс содержит три переменные типа (T, V, K), конструктор, принимающий на вход параметры типа (T, V, K), методы возвращающие значения трех переменных. Создать метод, выводящий на консоль имена классов для трех переменных класса. Наложить ограничения на параметры типа: T должен реализовать интерфейс `Comparable` (классы оболочки, `String`), V должен реализовать интерфейс `DataInput` и расширять класс `InputStream`, K должен расширять класс `Number`.

    Вариант решения

```
1   private static class MyClass<T extends Comparable, V extends
        InputStream & Serializable, K extends Number> {
2     T t; V v; K k;
3
4     MyClass(T t, V v, K k) {
5       this.t = t; this.v = v; this.k = k;
6     }
7
8     public T getT() { return t; }
9     public V getV() { return v; }
10    public K getK() { return k; }
11
```

```
12      void print() {
13         System.out.printf("t = %s, v = %s, k = %s",
14               t.getClass().getName(),
15               v.getClass().getName(),
16               k.getClass().getName());
17      }
18   }
```

Задание 2

- **Ценность этапа** Практика написания обобщённого кода с дополнительным описанием алгоритмов манипулирования данными.
- **Тайминг** 15-20 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
  - Если группа студентов справилась с заданием, а времени осталось более 5 минут, выдавать группе задания «со звёздочкой».
- **Задание**:
  - Описать собственную коллекцию – список на основе массива. Коллекция должна иметь возможность хранить любые типы данных, иметь методы добавления и удаления элементов.

    Вариант решения

```
1  private static class OwnList<T> {
2     Object[] arr;
3     int count;
4
5     OwnList() {
6        arr = new Object[1];
7        count = 0;
8     }
9
10    void add(T item) {
11       if (count == arr.length) {
12          Object[] newArr = new Object[arr.length * 2];
13          System.arraycopy(arr, 0, newArr, 0, arr.length);
14          arr = newArr;
15       }
16       arr[count++] = item;
17    }
18
19    T remove() {
20       if (count == 0) throw new NoSuchElementException();
21       T temp = (T) arr[--count];
22       arr[count] = null;
23       return temp;
24    }
25
```

```
26      @Override
27      public String toString() {
28         return Arrays.toString(arr);
29      }
30  }
```

Задание 3

- **Ценность этапа** Изучение поведения итератора для понимания механизмов его действия и необходимости его использования в работе.
- **Тайминг** 20-25 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
- **Задание**:
  - Написать итератор по массиву. Итератор – это объект, осуществляющий движение по коллекциям любого типа, содержащим любые типы данных. Итераторы обычно имеют только два метода – проверка на наличие следующего элемента и переход к следующему элементу. Но также, особенно в других языках программирования, возможно встретить итераторы, реализующие дополнительную логику.

    Вариант решения

```
1      private static class ArrayIterator<T> {
2
3         private final T[] array;
4         private int index = 0;
5
6         public ArrayIterator(T[] array) {
7            this.array = array;
8         }
9
10        public boolean hasNext() {
11           return index < array.length;
12        }
13
14        public T next() {
15           if(!hasNext())
16              throw new NoSuchElementException();
17           return array[index++];
18        }
19     }
20
21     public static void main(String[] args) {
22        Integer[] arr = {1,2,3,4};
23        ArrayIterator<Integer> it = new ArrayIterator<>(arr);
24        while (it.hasNext()) {
25           System.out.print(it.next() + " ");
26        }
27     }
```

*1 Написать итератор таким образом, чтобы его возможно было использовать для цикла foreach.

Вариант решения

Указать, что для этого достаточно реализовать интерфейс Iterator<T>.

```
class ArrayIterator<T> implements Iterator<T>{

   private T[] array;
   private int index = 0;

   public ArrayIterator(T[] array) {
      this.array = array;
   }

   @Override
   public boolean hasNext() {
      return index < array.length;
   }

   @Override
   public T next() {
      if(!hasNext())
         throw new NoSuchElementException();
      return array[index++];
   }
}
```

Задание 4

- **Ценность этапа** Закрепление понимания механизмов работы коллекций (в том числе с применением итераторов).
- **Тайминг** 20-25 мин
- **Действия преподавателя**
  - Выдать задание студентам;
  - Подробно объяснить, что именно требуется от студентов, избегая упоминания конкретных языковых конструкций;
- **Задание**:
  - Описать интерфейс Person с методами doWork() и haveRest(). Написать два класса работник и бездельник, реализующих интерфейс. Работник работает, и не умеет бездельничать, в то время как бездельник не умеет работать, но умеет отдыхать. Написать обобщённые классы Workplace и Club, содержащие массив из Person. В классах необходимо вызывать у всего массива людей вызывать соответствующие методы.

  Вариант решения

```
   interface Person {
      void doWork();
      void haveRest();
```

```
4    }
5
6    private static class Worker implements Person {
7       @Override public void doWork() {
8          System.out.println("Work!"); }
9       @Override public void haveRest() {
10         System.out.println("What?");
11      }
12   }
13
14   private static class Idler implements Person {
15      @Override public void doWork() {
16         System.out.println("No!"); }
17      @Override public void haveRest() {
18         System.out.println("Chill!");
19      }
20   }
21
22   private static class Workplace<T extends Person> {
23      Person[] arr;
24      public Workplace(T... people) {
25         arr = people;
26      }
27      void work() {
28         for (Person person : arr) { person.doWork(); }
29      }
30   }
31   private static class Club<T extends Person> {
32      Person[] arr;
33      public Club(T... people) {
34         arr = people;
35      }
36      void chill() {
37         for (Person person : arr) { person.haveRest(); }
38      }
39   }
40
41   public static void main(String[] args) {
42      Workplace<Person> w = new Workplace<>(new Worker(), new
           Worker(), new Idler());
43      Club<Person> c = new Club<>(new Worker(), new Worker(), new
           Idler());
44      w.work();
45      c.chill();
46   }
```

Домашнее задание

– **Ценность этапа** Задать задание для самостоятельного выполнения между занятиями.

- **Тайминг** 5-10 минут.
- **Действия преподавателя**
  - Пояснить студентам в каком виде выполнять и сдавать задания
  - Уточнить кто будет проверять работы (преподаватель или ревьювер)
  - Объяснить к кому обращаться за помощью и где искать подсказки
  - Объяснить где взять проект заготовки для дз
- **Задания**

5-25 мин Выполнить все задания семинара, если они не были решены, без ограничений по времени;

Все варианты решения приведены в тексте семинара выше

25 мин 1. Внедрить итератор из задания 2 в коллекцию, написанную в задании 3 таким образом, чтобы итератор был внутренним классом и, соответственно, объектом в коллекции.

```
private static class OList<T> implements Iterable<T> {
   class ArrayIterator<T> implements Iterator<T> {
      private int index = 0;

      @Override
      public boolean hasNext() {
         return index < count;
      }

      @Override
      public T next() {
         if(!hasNext())
            throw new NoSuchElementException();
         return (T) arr[index++];
      }
   }

   Object[] arr;
   int count;
   Main.OList.ArrayIterator iter;

   OList() {
      arr = new Object[1];
      count = 0;
      this.iter = new ArrayIterator<T>();
   }

   void add(T item) {
      if (count == arr.length) {
         Object[] newArr = new Object[arr.length * 2];
         System.arraycopy(arr, 0, newArr, 0, arr.length);
         arr = newArr;
      }
      arr[count++] = item;
   }
   T remove() {
      if (count == 0) throw new NoSuchElementException();
```

```
        --count;
        T temp = (T) arr[count];
        arr[count] = null;
        return temp;
    }

    @Override
    public Iterator<T> iterator() {
        return iter;
    }

    @Override
    public Spliterator<T> spliterator() {
        return Iterable.super.spliterator();
    }

    @Override
    public String toString() {
        return Arrays.toString(arr);
    }
}
```

15 мин 2. Написать класс Калькулятор (необобщенный), который содержит обобщенные статические методы: `sum()`, `multiply()`, `divide()`, `subtract()`. Параметры этих методов – два **числа** разного типа, над которыми должна быть произведена операция.

```
private static class Calculator {
    public static <T extends Number, U extends Number> double sum(T
        num1, U num2) {
        return num1.doubleValue() + num2.doubleValue();
    }

    public static <T extends Number, U extends Number> double
        multiply(T num1, U num2) {
        return num1.doubleValue() * num2.doubleValue();
    }

    public static <T extends Number, U extends Number> double divide(T
        num1, U num2) {
        return num1.doubleValue() / num2.doubleValue();
    }

    public static <T extends Number, U extends Number> double
        subtract(T num1, U num2) {
        return num1.doubleValue() - num2.doubleValue();
    }
}

public static void main(String[] args) {
    System.out.println(Calculator.sum(2, 2.0));
```

```
    System.out.println(Calculator.multiply(2.0f, 2.0));
    System.out.println(Calculator.divide(2L, 2.0));
    System.out.println(Calculator.subtract(2, 2));
}
```

10 мин 3. Напишите обобщенный метод `compareArrays()`, который принимает два массива и возвращает `true`, если они одинаковые, и `false` в противном случае. Массивы могут быть любого типа данных, но должны иметь одинаковую длину и содержать элементы одного типа.

```
private static <T> boolean compareArrays(T[] array1, T[] array2) {
    if (array1.length != array2.length) {
        return false;
    }

    for (int i = 0; i < array1.length; i++) {
        if (!array1[i].equals(array2[i])) {
            return false;
        }
    }

    return true;
}
```

10 мин 4. Напишите обобщенный класс `Pair`, который представляет собой пару значений разного типа. Класс должен иметь методы `getFirst()`, `getSecond()` для получения значений пары, а также переопределение метода `toString()`, возвращающее строковое представление пары.

```
private static class Pair<T1, T2> {
    private T1 first;
    private T2 second;

    public Pair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }

    public T1 getFirst() { return first; }
    public T2 getSecond() { return second; }

    @Override public String toString() {
        return "(" + first + ", " + second + ")";
    }
}

public static void main(String[] args) {
    Pair<Integer, String> pair1 = new Pair<>(1, "one");
    System.out.println(pair1.getFirst()); // 1
    System.out.println(pair1.getSecond()); // "one"
    System.out.println(pair1); // "(1, one)"
```

```
23
24      Pair<String, Double> pair2 = new Pair<>("pi", 3.14);
25      System.out.println(pair2.getFirst()); // "pi"
26      System.out.println(pair2.getSecond()); // 3.14
27      System.out.println(pair2); // "(pi, 3.14)"
28  }
```

Рефлексия и завершение семинара

- **Цель этапа:** Привести урок к логическому завершению, посмотреть что студентам удалось, что было сложно и над чем нужно еще поработать
- **Тайминг:** 5-10 минут
- **Действия преподавателя:**
  - Запросить обратную связь от студентов.
  - Подчеркните то, чему студенты научились на занятии.
  - Дайте рекомендации по решению заданий, если в этом есть необходимость
  - Дайте краткую обратную связь студентам.
  - Поделитесь ощущением от семинара.
  - Поблагодарите за проделанную работу.

Термины, определения и сокращения

`finally`	часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.
`throws`	ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.
`throw`	оператор, активирующий (выбрасывающий) объект исключения.
`try...catch`	двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.
BIOS	(англ. basic input/output system – «базовая система ввода-вывода») – набор микропрограмм, реализующих низкоуровневые API для работы с аппаратным обеспечением компьютера, а также создающих необходимую программную среду для запуска операционной системы у IBM PC-совместимых компьютеров.
CI	(англ. Continious Integration) практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.
CLI	(англ. Command line interface, Интерфейс командной строки) — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».
cp1251	набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 10-й версии. Была создана на базе кодировок, использовавшихся в ранних русификаторах Windows.
Docker	программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут почти на любой системе.

FAT	(англ. File Allocation Table «таблица размещения файлов») – классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей.
GPL	GNU General Public License (чаще всего переводят как Открытое лицензионное соглашение GNU) — лицензия на свободное программное обеспечение, созданная в рамках проекта GNU, по которой автор передаёт программное обеспечение в общественную собственность. Её также сокращённо называют GNU GPL или даже просто GPL, если из контекста понятно, что речь идёт именно о данной лицензии. GNU Lesser General Public License (LGPL) — это ослабленная версия GPL, предназначенная для некоторых библиотек ПО.
IDE	(от англ. Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
JDK	(от англ. Java Development Kit) — комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов, примеры, документацию, различные утилиты и исполнительную систему. В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.
JIT	(англ. Just-in-Time, компиляция «точно в нужное время»), динамическая компиляция — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.
JRE	(от англ. Java Runtime Environment) — минимальная (без компилятора и других средств разработки) реализация виртуальной машины, необходимая для исполнения Java-приложений. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов.
JVM	(от англ. Java Virtual Machine) — виртуальная машина Java, основная часть исполняющей системы Java. Виртуальная машина Java исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором. JVM может также использоваться для выполнения программ, написанных на других языках программирования.
NTFS	(аббревиатура от англ. new technology file system – «файловая система новой технологии») – стандартная файловая система для семейства операционных систем Windows NT фирмы Microsoft.

SDK	(от англ. software development kit, комплект для разработки программного обеспечения) — это набор инструментов для разработки программного обеспечения в одном устанавливаемом пакете. Они облегчают создание приложений, имея компилятор, отладчик и иногда программную среду. В основном они зависят от комбинации аппаратной платформы компьютера и операционной системы.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
String	Класс в Java, предназначенный для работы со строками. Все строковые литералы, определенные в Java программе – это экземпляры класса String
Swing	библиотека для создания графического интерфейса для программ на языке Java. Swing разработан компанией Sun Microsystems и содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и тому подобные.
Typecasting	преобразование типов переменных в типизированных языках программирования
UTF-8	(от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
Асинхронность	вид программирования, позволяющий вынести выполняемые задачи отдельными блоками кода. Применяется в сервисах, где предыдущее действие тормозит следующее. Синхронный процесс выполняется поэтапно. Пользователь совершает действие, ждёт, пока программа обработает часть кода, переходит к другому блоку. При использовании асинхронности, код убирает операцию, блокирующую следующие действия.
Ввод-вывод	взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек (субъект), так и любая другая система обработки информации
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Загрузчик	системное программное обеспечение, обеспечивающее загрузку операционной системы непосредственно после включения компьютера (процедуры POST) и начальной загрузки.
Идентификатор	идентификатор переменной - название переменной, по которому возможно получить доступ к области памяти, соответствующей этой переменной

Инициализация	одновременной объявление переменной и присваивание ей значения
Инкапсуляция	(англ. encapsulation, от лат. in capsula) — в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Класс	определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Определяет новый тип данных
Компонент	независимый модуль программы, предназначенный для повторного использования. Часто компоненты объединяют по общим признакам и организовывают в соответствии с определёнными правилами и ограничениями. Например, компоненты графического интерфейса.
Компоновщик	Компоновщик (layout manager) в библиотеке Java Swing отвечает за расположение и организацию компонентов (например, кнопок, текстовых полей, панелей). Он определяет, как компоненты будут выравниваться, размещаться и изменяться при изменении размеров окна.
Конструктор	это частный случай метода, предназначенный для инициализации объектов при создании в Java.
Куча	адресуемое пространство оперативной памяти компьютера. Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Массив	структура данных, хранящая набор значений в непрерывной области памяти
Метод	функция в языке программирования, принадлежащая классу
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.

Наследование	(англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
ОС	(операционная система) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.
Обработчик	это механизм, с помощью которого приложение может перехватить события, такие как сообщения, действия мыши и нажатия клавиш. Функция, которая перехватывает события определенного типа, называется процедурой-обработчиком. Процедура-обработчик может действовать для каждого получаемого события, а затем изменить или отменить событие.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.
Объект	конкретный экземпляр класса, созданный в программе
Окно	это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя – активное окно. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно.
ПО	программное обеспечение
Панель	Панель в библиотеке Java Swing представляет собой контейнер, который используется для группировки и организации других компонентов в пользовательском интерфейсе. Она представляет собой область с фиксированным размером на которую можно добавить другие компоненты, такие как кнопки, текстовые поля или изображения.
Параллельность	способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно. Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).
Переполнение	целочи́сленное переполне́ние (англ. integer overflow) — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в тип данных

Перечисление — это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.

Подавленное искл. — исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.

Полиморфизм — это возможность объектов с одинаковой спецификацией иметь различную реализацию

Потоки в-в — объекты, из которых можно считать данные и в которые можно записывать данные

Разделы — (англ. partition), раздел диска (англ. disk partition) – часть долговременной памяти накопителя данных (жёсткого диска, SSD, USB-накопителя), логически выделенная для удобства работы, и состоящая из смежных блоков.

Сборщик мусора — специализированная подпрограмма, очищающая память от неиспользуемых объектов.

События — это действия или случаи, возникающие в программируемой системе, о которых система сообщает для того, чтобы было возможно с ними взаимодействовать. Например, если пользователь нажимает кнопку на графическом интерфейсе, возможно ответить на это действие, отобразив информационное окно.

Статика — (статический контекст) static - (от греч. неподвижный) — раздел механики, в котором изучаются условия равновесия механических систем под действием приложенных к ним сил и возникших моментов. В языке программирования Java - принадлежность поля и его значения не объекту, а классу, и, как следствие, доступность такого поля и его значения в единственном экземпляре всем объектам класса.

Стек — структура данных, работающая по принципу LIFO (last in, first out) или FILO (first in, last out). Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.

Строка — ряд знаков, написанных или напечатанных в одну линию. Строка может также означать строковый тип данных в программировании.

Типизация — классификация по типам

ФС — Файловая система (File System) – один из ключевых компонентов всех операционных систем. В ее обязанности входит структуризация, чтение, хранение, запись файловой документации. Она напрямую влияет на физическое и логическое строение данных, особенности их формирования, управления, допустимый объем файла, количество символов в его названии и прочие.

Файл	именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.