Содержание

3. Специализация: ООП

3.1. 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).

3.2. 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 ;

3.3. Классы и объекты, поля и методы, статика

3.3.1. 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.

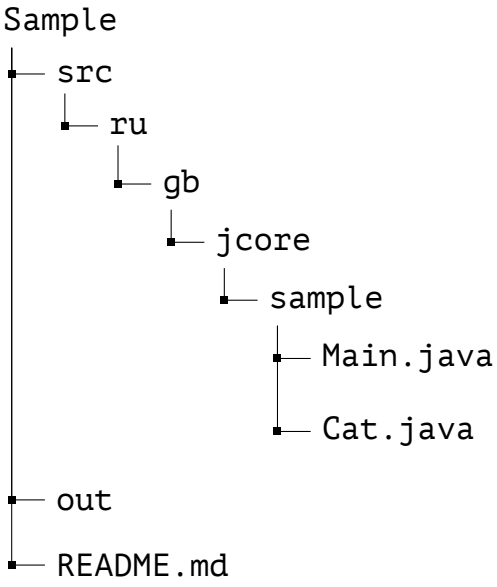

Рис. 1: Структура проекта

3.3.2. Поля класса

Начнем описывать в классе `Cat` так называемый API кота. Как известно, 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. Свойства класса, записанные таким образом, в виде переменных, называются **полями**.

Листинг 1: Структура кота в программе

```
1  package ru.gb.jcore;
2
3  public class Cat {
4      String name;
5      String color;
6      int age;
7  }
```

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).

```
1  Cat cat0; // cat0 = null;
2  cat0 = new Cat();
3  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. Рас-

смотрим 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.

```
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);
      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. 2.

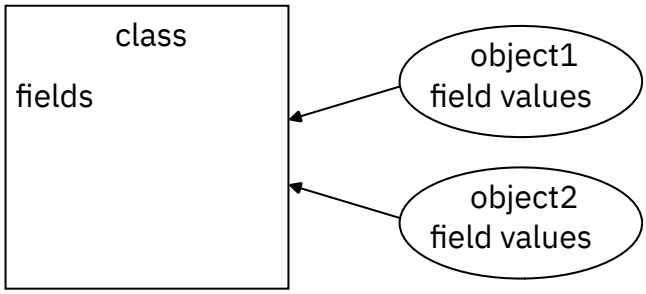

Рис. 2: Логическая структура отношения класс-объект

3.3.3. 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.

```
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.

3.3.4. 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.

```
Cat cat1 = new Cat();
Cat cat2 = cat1;

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);
```

```
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.

3.3.5. 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. 3.

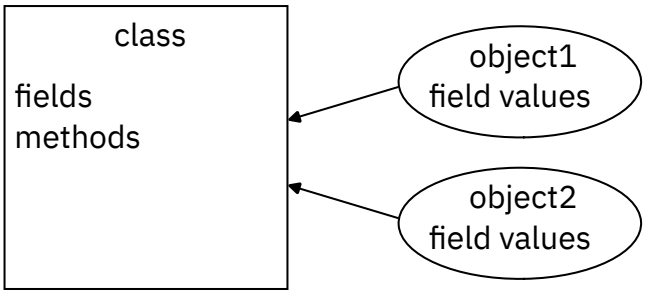

Рис. 3: Логическая структура отношения класс-объект

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.

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

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

   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.

3.3.6. 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.

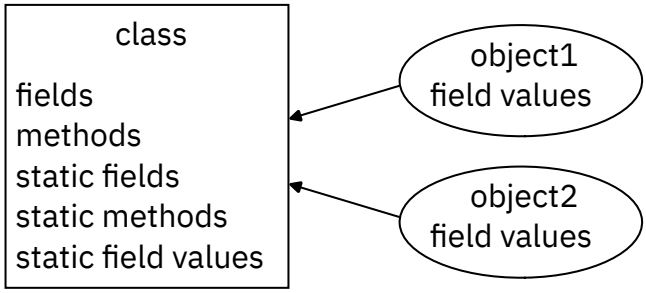

Рис. 4: Логическая структура отношения класс-объект

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 4 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 ещ§ и опасный инструмент создания общих свойств. Так, например, мы знаем, что у котов четыре лапы, а не 6 и не 8. Не создавая никакого барсика будет понятно, что у кота - 4 лапы. Это полезное поведение.

лайвкод 03-статическое-поле-ошибка Посмотрим на опасность. Мы видим, что у каждого кота есть имя, и помним, что коты хранят значение своего имени каждый сам у себя. А знают экземпляры о названии поля потому что знают, какого класса они экземпляры. Но что если мы по невнимательности добавим свойство статичности к имени кота?

03-статическое-поле-признак Создав тех же самых котов, которых мы создавали весь урок, мы получим двух мурзиков и ни одного барсика. Почему это произошло? По факту переменная у нас одна на всех, и значение тоже одно, а значит каждый раз мы меняем именно его, а все остальные коты ничего не подозревая смотрят на значение общей переменной и бодро его возвращают. Поэтому, чтобы не запутаться, к статическим переменным, как правило, обращаются не по ссылке на объект — cat1.name, а по имени класса — Cat.name.

03-статические-поля К слову, 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.

3.3.7. 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.

3.4. 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 5 представлена общая модель организации памяти в Java.

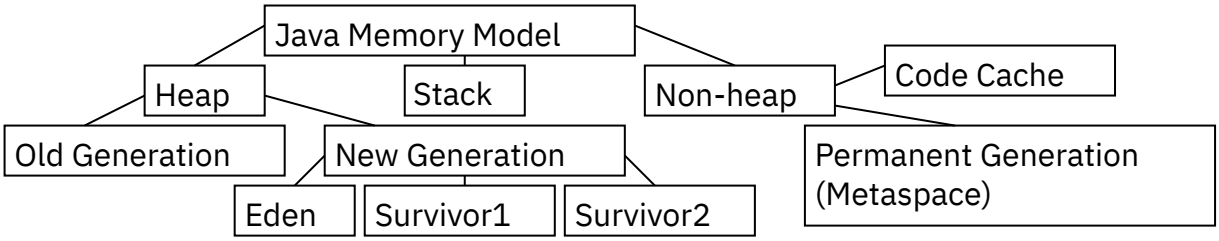

Рис. 5: Устройство памяти

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 хра-

нится 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

3.4.1. Задания для самопроверки

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

3.5. Конструкторы

3.5.1. 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 2, 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 2: Не самый лучший конструктор

```
1  public class Cat {
2     String name;
3     String color;
4     int age;
5
6     public Cat() {
7        name = "Barsik";
8        color = "White";
9        age = 2;
10    }
```

```
11
12      // ...
13  }
```

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 3: Параметризированный конструктор

```
1  public class Cat {
2      String name;
3      String color;
4      int age;
5
6      Cat(String n, String c, int a) {
7          name = n;
8          color = c;
9          age = a;
10     }
11
12     // ...
13 }
```

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.

```
1  Cat cat1 = new Cat("Barsik", "White", 4);
2  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.

3.5.2. 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 3), 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 4: Использование ключевого слова `this` для параметров

```
1  public class Cat {
2     String name;
3     String color;
4     int age;
5
6     public Cat(String name, String color, int age) {
7        this.name = name;
8        this.color = color;
9        this.age = age;
10    }
11
12    // ...
13 }
```

Ключевое слово `this` сошлётся на вызвавший объект, в результате чего (в листинге 4)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 кон-

структоре ранее[1]. В листинге 4 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 5: Использование ключевого слова `this` для конструктора

```
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.

```
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, 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

[1] 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.

3.5.3. 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?

3.6. 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.

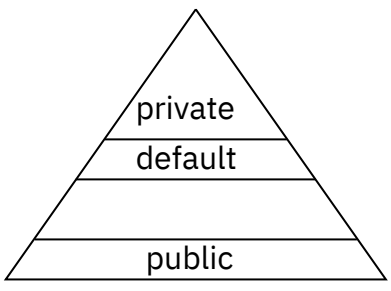

Рис. 6: Модификаторы доступа и их относительная область видимости

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

Внимательно рассмотрим класс котика из листинга 4. 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 6: 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;
   }
}
```

В листинге 6 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 ока-

заться 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.

3.6.1. Задания для самопроверки

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

3.7. Наследование

3.7.1. 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 7: 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 8: 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 9), 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 9: 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 10 и 11. 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 переменную 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 10: 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 11: 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.

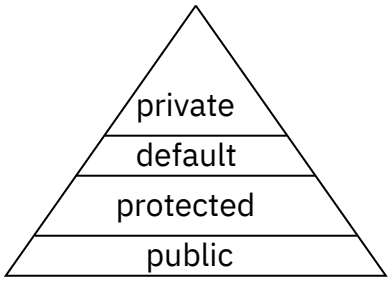

Рис. 7: Модификаторы доступа и их относительная область видимости

То есть верным вариантом в листинге 9 будет применение модификатора `protected`.

3.7.2. 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 12: Класс животного

```
1  public class Animal {
2     private String name;
3     private String color;
4     private int age;
5
6     public Animal(String name, String color, int
          age) {
7        this.name = name;
8        this.color = color;
9        this.age = age;
10    }
11
12    public String getName() {
13      return name;
14    }
15
16    public String getColor() {
17      return color;
18    }
19
20    public int getAge() {
21      return age;
22    }
23
24    public void setName(String name) {
25      this.name = name;
26    }
27
28    public void setColor(String color) {
29      this.color = color;
30    }
31
32    public void setAge(int age) {
33      this.age = age;
34    }
35
36    void move() {
```

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

Листинг 13: Класс кота

```
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 14: Класс собаки

```
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 15: Класс птицы

```
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.

3.7.3. `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`/

3.7.4. 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. 8: Ошибка наследования от final класса

3.7.5. 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.

3.7.6. 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

3.8. 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 16: Класс кота

```
1 public abstract class Animal {
2     // ...
3
4     void voice();
5
6     // ...
7 }
8
```

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

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

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 17: Класс кота

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

   @Override
   void move() {
      System.out.println(name + " crawls");
   }

   @Override
   void voice() {
      System.out.println(name + " hisses");
   }
}
```

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.

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

3.8.1. 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.