Содержание

5. Специализация: тонкости работы

5.1. 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.

5.2. 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;

5.3. 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.

5.3.1. 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.

5.3.2. 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 картоте-

ка: 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.

5.3.3. Windows

Линейка файловых систем для Windows: какую роль они играют в работе системы и как они развивались.

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.

5.3.4. 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 преимуществ 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 вхо-

дит 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.

5.3.5. 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?

5.4. Файловая система и представление данных

5.4.1. 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: Создание объекта файла

```
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 2: Использования файла для директории

```
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 3: Методы класса `File`

```
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`.

5.4.2. 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 – Unifrom 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 4: 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 5: 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 6: 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 7: 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);
   }
}
```

5.4.3. Задание для самопроверки

Ссылка на местонахождение – это:

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

5.5. 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 их более 4 типов), или передавать числа в различных представлениях. Доступ к данным может потребоваться как последовательный, 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.

5.5.1. 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.

5.5.2. 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 8: Использование Byte Array Stream

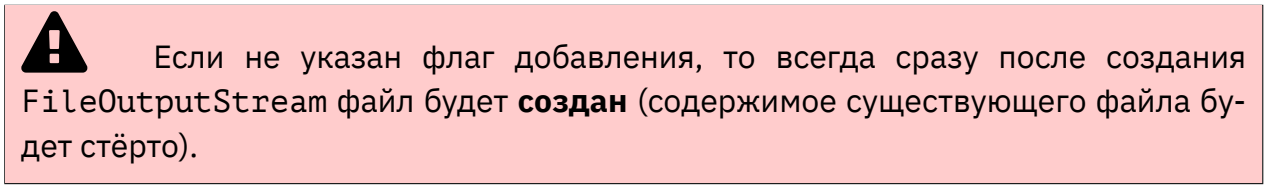
```
ByteArrayOutputStream out = new ByteArrayOutputStream();

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);
```

5.5.3. 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.



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.

5.5.4. 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 разу 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.

5.5.5. `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 10 миллионов символов.

Листинг 9: 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 10: 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 11: 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
```

5.5.6. 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 12: 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 13: 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 14: 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());
```

```
10
11      inData.close();
12  } catch (Exception e) {
13      System.out.println("Impossible IOException occurs: " + e.toString());
14      e.printStackTrace();
15  }
```

```
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.

5.5.7. 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.

5.5.8. 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`?

5.6. `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.

5.6.1. 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 15: 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.

5.7. 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.

5.7.1. 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

5.7.2. 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.

5.7.3. 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 (не разбираться с кодировками) внутри файлов следует писать текст только латинскими буквами.
2. написать метод, осуществляющий конкатенацию переданных ей в качестве параметров файлов (не особо важно, в первый допишется второй или во второй первый, или файлы вовсе объединятся в какой-то третий);
3. написать метод поиска слова внутри файла.