Конспект спринта №6

Вся правда о строках

Особенности применения строк

Любое сообщение в браузере, текст для ввода и вывода, время, даты — всё это строки. Ранее строка состояла из массива символов `char value[]`, но в Java 9 её заменили на

массив байтов `byte value[]`. Использование массива байтов вместо массива символов позволило выделять на строку ровно столько места, сколько требуется. Но на написание кода это никак не повлияло.

Подстрока — это непрерывный набор символов внутри строки. Строка также всегда будет подстрокой и для самой себя. Подстроки чувствительны к регистру.

Ориентироваться внутри строк и искать подстроки позволяет внутренняя нумерация символов. Она ничем не отличается от нумерации в массиве и тоже начинается с нуля.

Поскольку строки являются объектами, их можно создавать через ключевое слово `new`, но тогда JVM не сможет оптимизировать код и занести её в особое хранилище — **пул строк**. Если строка находится в пуле, и вы попытаетесь создать строку с таким же значением, то она не будет создаваться второй раз. Вместо этого вернётся ссылка на уже существующую. По-возможности, всегда создавайте строки через литерал:

```
String name = "Байт"; // "Байт" — строковый литерал
```

Вместо оператора `==` для строк нужно всегда использовать `equals()` поскольку так происходит сравнение значений, а не ссылок. Этот метод вместе с `toString()` и `hashCode()` уже корректно переопределены для `String` — как одного из основных классов стандартной библиотеки.

Полезные методы `String`

Класс `String` содержит несколько десятков методов. Самые популярные делятся на две категории:

Работа со строками

- `length()` узнаёт, сколько символов в строке.
- `isEmpty()` возвращает ответ, является ли строка пустой или нет.
- `isBlank()` вернёт `true`, если строка действительно пустая (`""`) или содержит только пробельные символы.
- `trim()` возвращает строку, из которой удалены пробельные символы в начале и в конце.
- `toUpperCase()` и `toLowerCase()` для перевода строки в верхний и нижний регистры.
- `charAt(int index)` возвращает элемент по индексу. Символ по индексу обязательно должен существовать, иначе программа выбросит исключение.

- Метод `split(String regex)` превращает строку в массив строк `String[]`. Элементы в строке разделяются по разделителю `regex`, который передаётся в метод.

> 💡 Метод `split(String regex)` принимает в качестве аргумента не просто символ, а **регулярное выражение** (`regex` — сокращение от англ. *regular expression*). Регулярные выражения — это особые строки из обычных и специальных символов, которые используются для поиска.

- `join(String delimeter, String... strs)` собирает в строку массив.

> 💡 Обратите внимание на запись второго аргумента в методе `join()` — `String... strs`. Такая запись (varargs) означает, что метод может принять любое количество аргументов. Поэтому специально создавать массив для метода `join()` необязательно.

Работа с подстроками

- `indexOf(String str)` и `lastIndexOf(String str)` позволяет найти индекс начала подстроки. Разница методов в том, что `indexOf` ищет слева направо, а `lastIndexOf` справа налево. Если подстрока не найдена, вернётся `-1`. Эти методы также могут принимать в качестве второго аргумента индекс, с которого нужно начать поиск.
- `boolean contains(String other)` возвращает `true`, если подстрока будет обнаружена **в любом месте** строки, иначе `false`.
- `boolean startsWith(String other)` вернёт `true`, если подстрока находится **в начале** строки, иначе `false`.
- `boolean endsWith(String other)` вернёт `true`, если подстрока **в конце** строки, иначе `false`.
- `String replace(String target, String replacement)` подменяет все вхождения подстроки `target` на строку `replacement`.
- `String replaceFirst(String target, String replacement)` заменяет первое вхождение искомой подстроки.
- `String substring(int beginIndex)`: если передать в метод один индекс `beginIndex`, то он вернёт все символы, начиная от этого индекса и до самого конца строки. Второй вариант — передать в `substring()` два индекса: не только начальный `beginIndex`, но и конечный `endIndex`. В этом случае метод вернёт подстроку, начиная от символа с

`beginIndex` и заканчивая символом с индексом `endIndex-1`. Символ с индексом `endIndex` исключается из результата.

При использовании методов замены для строк всегда создаётся новый объект, а не меняется старый, потому что у строк есть одно важное свойство — **свойство неизменяемости**. То есть поменять значение отдельных символов внутри строки после инициализации не получится.

StringBuilder

Из-за неизменяемости строк цикличные операции, вроде поиска подстроки и её удаления, при работе с текстами большого размера приведут к созданию большого количества лишних объектов. `StringBuilder` позволяет менять содержание строки без создания новой.

В отличие от `String` класс `StringBuilder` представляет строку в виде изменяемого набора символов. `StringBuilder` — обычный класс, поэтому его объекты создаются через ключевое слово `new`. Объекты `StringBuilder` редко используются как независимые сущности. Их основная задача — помочь в создании строки. После работы с ними их всегда можно привести к неизменяемой строке `String` с помощью метода `toString()`.

Внутренняя реализация `StringBuilder` отдалённо похожа на реализацию списка `ArrayList<T>`. При создании экземпляра `StringBuilder` выделяется внутренний буфер определённой **вместимости**, где и будут храниться элементы. Этот буфер может увеличиваться в размере по необходимости. Кроме того, вместимость можно задать сразу при создании `StringBuilder` при помощи конструктора, принимающего `int`.

```
// Создаём объект StringBuilder с начальной вместимостью в 100 элементов
StringBuilder sb = new StringBuilder(100);
```

Если начальное значение строки не задано, то будет создан пустой объект `StringBuilder`.

Методы `StringBuilder`

Для работы со `StringBuilder` предусмотрено множество методов. Вот некоторые из них:

Сигнатура	Описание
`indexOf(String str)` и `lastIndexOf(String str)`	Возвращают индекс элемента в `StringBuilder`.
`substring(int beginIndex)` и `substring(int beginIndex, int endIndex)`	Берут подстроку от `StringBuilder`. Возвращают объект класса `String`.
`append(String another)`	Добавляет строку-аргумент в конец `StringBuilder`.
`insert(int index, String str)`	Вставляет строку `str`, начиная с позиции `index`.

Сигнатура	Описание
`replace(int from, int to, String replacement)`	Заменяет подстроку с индекса `from` включительно до `to` не включительно на замену `replacement`.
`deleteCharAt(int index)`	Удаляет символ из `StringBuilder` по индексу.
`delete(int from, int to)`	Удаляет все символы с индекса `from` включительно до `to` не включительно.
`reverse()`	Отзеркаливает текст.
`setLength(int newLength)`	Устанавливает длину для `StringBuilder`. Все лишние символы будут отсечены.

Полный список можно посмотреть в документации.

> 💡 Выработайте привычку перед тем, как реализовывать некую базовую функциональность, проверять, нет ли её в классах стандартной библиотеки. В худшем случае вы убедитесь, что придётся самим писать метод, в лучшем — быстро найдёте нужный в стандартной библиотеке.

Форматирование строк

Чтобы строки выводились не вразнобой, а в нужном порядке, требуется их предварительно форматировать. Для форматирования строк используется статический метод `String.format(String format, Object... args)`. В него передаётся два аргумента.

- Первый — образец для форматирования `String format`, это строка, в которой, как правило, используются **символы преобразования** (англ. «conversion character»), хотя их может и не быть.
- Второй — varargs `Object... args` — подразумевается, что он должен содержать столько аргументов, сколько в `String format` символов преобразования.

```
//Без форматирования
String trafficLight = "Цвета светофора: " + colors[0] + ", " + colors[1] + " и " + colors[2] + "."

// С форматированием
String trafficLight = String.format("Цвета светофора: %s, %s и %s.", colors[0], colors[1], colors[2]);
```

Символы преобразования

Для разных типов данных нужны разные символы преобразования. Они состоят из знака процентов `%` и обязательного указания типа данных.

- `%s` — для строк (*s* — сокращение от *string*);

- `%d` — для целых чисел (*d* от *decimal*);
- `%f` — для чисел с плавающей точкой (*f* от *float*);
- `%b` — для булевых значений (*b* от *boolean);*
- `%c` — для символов (*c* от *char* и *character*).

Если понадобятся другие, менее распространённые, их можно найти в документации.

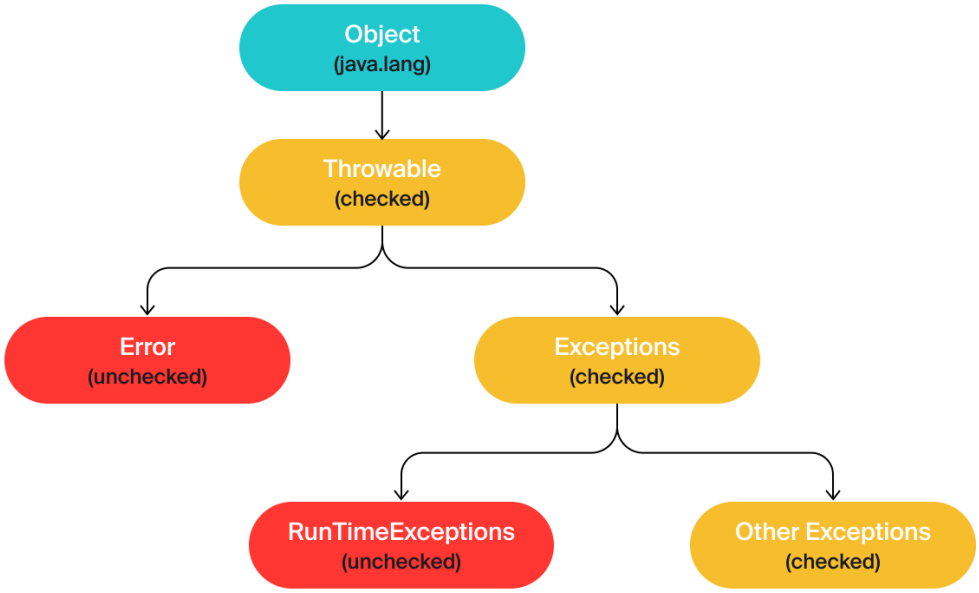
Дополнительные параметры символов преобразования

- Чтобы вывести строку в верхнем регистре, вместо символа `%s` понадобится `%S`.
- Задать нужное расположение можно с помощью унификации длины строки. Для этого нужно добавить положительное число между знаком `%` и обозначением `s`. Это число обозначает минимальное число символов. В результате строка станет нужной длины — недостающие символы заполнятся пробелами. Чтобы выровнять текст по левой стороне, целое число после `%` нужно сделать отрицательным.
- Чтобы ограничить количество символов в строке, нужно задать её диапазон. Это происходит с помощью **параметра точности**. Точность задаётся вежду `%` и `s` в виде числа с плавающей точкой. Целая часть — минимальное количество символов, дробная — максимальное. Аналогичным образом можно задать количество символов после точки в дробях. Первое число в записи преобразования всё также будет обозначать минимальное количество выводимых символов, а вот второе уже количество символов после запятой. Округление при этом произойдёт по правилам математики.
- Добавлять переносы в форматируемую строку можно с помощью форматного символа `%n` или специального символа `\n`. В обоих случаях будет выполнен перенос строки. Отличие в том, что `%n` используется только в форматируемых строках, в то время как `\n` работает и в обычных тоже.

Методы `System.out.printft` и `String.format` не занимаются преобразованиями непосредственно. Они всего лишь передают форматируемую строку и аргументы в специальный класс `Formatter`. Именно он производит подстановку и преобразование аргументов. Использовать этот класс напрямую, как правило, не требуется, но иногда это бывает полезно. Например, в `Formatter` можно передать объект класса `StringBuilder`. Более подробно изучить этот вопрос можно в документации.

Исключения

Исключения — это те ошибки и сбои, которые возникают, когда программа уже запустилась, начала работать, но вдруг что-то пошло не так. Все исключения наследуют от одного класса — `Throwable`.



Группа `Exception` охватывает основную часть исключений. В неё входят ошибки при работе с файлами и сетью, сбои взаимодействия с базами данных и другие. Кроме того, от `Exception` идёт ветвь исключений с особым поведением — `RuntimeException`. К ним относятся выход за пределы массива или неверно переданные данные в метод, ошибки при арифметических операциях (например, деление на ноль) и обращения к неинициализированным объектам.

Группа `Error` — это классы, которые описывают критические ошибки, которые в большинстве случаев создают аварийную ситуацию и требуют перезапуска программы. Именно в результате исключений `Error` пользователь сталкивается с сообщением типа «Извините, произошла непредвиденная ошибка!». Это могут быть сбои из-за нехватки памяти — `OutOfMemoryError` (от англ. out of memory — «за пределами памяти») или проблемы с JVM — `VirtualMachineError`.

Все исключения делятся на два типа: **проверяемые (**обработка которых обязательна и является частью логики приложения) и непроверяемые (ошибки и нештатные ситуации, которые не были предусмотрены логикой программы и не обработаны разработчиками заранее). В большинстве случаев при непроверяемых исключениях работа программы немедленно прекращается. А без обработки проверяемых исключений программа даже не скомпилируется.

Определить, к какому из типов относится то или иное исключение, можно либо по иерархии классов, либо по их поведению в коде.

Типов исключений очень много, и поэтому иерархия исключений фактически выглядит примерно так:

- Непроверяемые исключения, которые наследуют от `Error`

`VirtualMachineError`	Базовые ошибки JVM, связанные с тем, что исчерпаны ресурсы или обнаружены повреждения.
`OutOfMemoryError`	Производный класс от `VirtualMachineError`, который показывает ошибки из-за нехватки памяти.
`StackOverflowError`	Производный класс от `VirtualMachineError`, показывает переполнение стека из-за того, что метод слишком много раз вызывал сам себя.
`AssertionError`	Ошибка утверждения.
`IOError`	Исключение, которое происходит при серьёзных ошибках ввода-вывода.
`ThreadDeath`	Возникает при вызове метода `Thread.stop()` у потока.

- Проверяемые исключения, которые наследуют от `Exception`

`IOException`	Это базовый класс проверяемых исключений ввода-вывода.
`EOFException`	Исключение, которое сигнализирует о внезапном достижении конца файла или потока.
`FileNotFoundException`	Файл по указанному пути не найден.
`FileSystemException`	Базовый класс для исключений файловой системы, таких как ошибка доступа, попытка создать уже существующий файл или удалить не пустую директорию.
`MalformedURLException`	Неверный синтаксис при создании класса ссылки из строки.
`SocketException`	Ошибки при создании/обрыве соединения по сокету.
`UnknownHostException`	Невозможность определить IP-адрес узла по доменному имени.
`SQLException`	Ошибки при работе с базой данных.
`TimeoutException`	Исключение, которое происходит у методов, выполнение которых ограничено по времени.

`URISyntaxException`	Неверный формат универсального идентификатора ресурсов.

- Непроверяемые исключения, которые унаследованы от `RuntimeException`.

`ArithmeticException`	Исключения при арифметических операциях. Например, деление на 0.
`IllegalArgumentException`	Возникает при неверно переданных в метод или конструктор параметрах.
`IndexOutOfBoundsException`	Исключение при выходе за заданный диапазон.
`NoSuchElementException`	Данного элемента больше не существует в перечислении.
`NullPointerException`	Приложение пытается использовать `null` в том месте, где требуется инициализированный объект.
`UnsupportedOperationException`	Операция не поддерживается вызываемым объектом.
`ClassCastException`	Невозможность привести объект к заданному типу.

Stack trace

Последовательность действий при выполнении программы обрабатывается в недрах JVM в виде стека. Это структура данных, которая реализована по принципу LIFO: элемент, который добавлен в стек последним, будет взят первым. Каждый вызов метода при выполнении программы записывается как элемент класса `StackTraceElement`. Когда выполнение метода завершается, информация о нём удаляется из стека.

Исключение прерывает работу одного или нескольких методов — они не могут завершиться и информация о них остаётся в памяти JVM в виде массива `StackTraceElement[]`. Сообщение об ошибке, стек-трейс, открывает доступ к этому массиву.

В первой строке стек-трейса сообщается базовая информация. Затем в обратном порядке (так как мы имеем дело со стеком) идёт история вызовов методов, при этом указывается класс каждого метода, его файл и местоположение в нём.

Конструкция `try`-`catch`

Для обработки исключений существует специальная конструкция из ключевых слов `try` (англ. «пробовать, пытаться») и `catch` (англ. «ловить»). В `try` передаётся код, из-за которого может произойти исключение. В `catch` в качестве аргумента — тип ошибки, а в тело — код, который выполнится, если произойдёт исключение, указанное в `try`. В коде `try`-`catch` выглядит так:

```
try {
    ... // код, из-за которого может произойти исключение
} catch (Throwable throwable) { // параметр - тип возможного иключения
    // код, который выполнится, если произойдёт исключение указанного типа
}
```

Обработка исключения настраивается исходя из логики программы.

Чтобы обрабатывать разные типы исключений, которые может сгенерировать метод или конструктор, нужно добавить несколько блоков `catch`. Каждый из них будет отвечать за конкретное исключение и выдавать соответствующий ответ для него. Часто требуется совместить два типа исключений, если инструкции для них являются одинаковым. Сделать это можно в блоке `catch` с помощью символа `|`.

При обработке исключений важно соблюдать правильную последовательность. Сначала обрабатываются исключения в классах-наследниках, и только потом родительские.

> 💡 Проверить цепочку наследования исключений проще всего в IDEA — для этого нужно воспользоваться либо комбинацией `Ctrl + левая кнопка мыши` (`Cmd + левая кнопка мыши` для Mac OS) либо сочетанием клавиш `Ctrl+B` (`Cmd+B` для Mac OS).

Методы Trowable

- Метод `printStackTrace()` — самый простой способ получить сразу всю информацию об ошибке.
- Метод `getMessage()` — получить короткое сообщение с описанием ошибки, например, такое — `/ by zero`. Использовать вызов `getMessage()` можно, только если при генерации исключений были применены конструкторы `Throwable` с параметром `String message`.
- Метод `getStackTrace()` возвращает массив элементов, представленных классом `StackTraceElement`.

В `StackTraceElement` обычно используется несколько методов:

- `getClassName()` — возвращает название класса, где произошло исключение,
- `getMethodName()` — позволяет получить имя вызванного при этом метода,
- `getFileName()` — подскажет имя файла,
- `getLineNumber()` — отобразит номер линии в файле.

Есть и другие методы с похожей функциональностью — их можно посмотреть в документации или увидеть в подсказке IDEA.

Связка `getStackTrace()` и методов `StackTraceElement` работает так. После того, как метод `getStackTrace()` возвращает массив данных об исключении, можно пройтись по нему циклом, чтобы отобразить нужную информацию с помощью методов `StackTraceElement`.

Как написать своё исключение

Класс исключения в обязательном порядке должен быть наследником `Throwable` или его потомков.

```
public class InputException extends Exception {
}
```

У суперкласса всех исключений `Throwable` четыре публичных конструктора:

- `Throwable()` — без параметров,
- `Throwable(String message)` — с передачей короткого описания ошибки,
- `Throwable(String message, Throwable cause)` — с передачей текста об ошибке и информации о её причине,
- `Throwable(Throwable cause)` — только с информацией о причине.

Их нужно переопределить, чтобы использовать в классе исключения.

Ключевое слово `throw` (англ. «бросать, выбрасывать») — позволяет сгенерировать экземпляр класса исключения с нужными параметрами.

```
throw new UserInputException("Ошибка ввода!"); // сгенерировали исключение
```

Если с помощью `throw` генерируется проверяемое исключение, то нужно сразу настроить его обработку. Это будет выглядеть так.

```
try {
    if (...) { // если возникла определённая ситуация
        throw new ThrowableClass(parameters); // сгенерировали исключение
    }
}
catch (ThrowableClass e) {
    // настроили обработку исключения, сгенерированного в try
}
```

При генерации исключения можно использовать любой доступный конструктор класса. Удобнее всего передавать в качестве параметра сообщение об ошибке, которое `catch` потом сможет вернуть с помощью метода `getMessage()`.

> 💡 Не стоит забывать, что исключение — это обычный класс, в него можно добавлять любые другие методы, поля и конструкторы.

Ключевое слово Throws

Ключевое слово `throws` обозначает, что метод генерирует исключение — то есть внутри него как раз может быть использовано `throw`. В первую очередь `throws` используется для указания проверяемых исключений, так как их обработка обязательна и является частью логики приложения. Когда мы вызываем метод, в котором есть указание на проверяемое исключение с помощью `throws`, то у нас появляется два варианта, как действовать дальше.

1. Обернуть этот метод в `try...catch` и добавить логику по обработке или выводу информации об ошибке.
2. Переадресовать обработку исключения вызывающему методу или JVM. Для этого нужно добавить `throws` с указанием исключения, которое может произойти, в тот метод, который вызывает метод с `throws`. В этом случае обработку исключения возьмёт на себя вызывающий метод или же JVM, если это точка запуска программы.

Если проверяемые исключение всегда должны быть отмечены с помощью `throws`, то для непроверяемых исключений это необязательно, так как они представляют собой ситуацию «у нас всё сломалось», которую невозможно предугадать. Однако указание непроверяемых исключений с помощью `throws` является хорошим тоном, потому что помогает сразу определить их. `throws` указывается после круглых скобок метода, который может выбросить исключение. После него через пробел идёт имя класса этого исключения.

```
public void methodWithException() throws FirstException {
     // какой-то код
    if (какое-то условие) {
        throw new FirstException(); // сгенерировать исключение
    }
    // какой-то код
}
```

Такая запись в коде означает, что метод `methodWithException()` может сгенерировать исключение `FirstException`.

Конструктор класса — это тоже по сути метод, который инициализирует класс и вызывает методы, которые могут сгенерировать исключение. Поэтому `throws` может использоваться и в конструкторах.

Принцип работы `finally`

Блок `finally` нужен при взаимодействии с объектами, которые требуют закрытия после того, как работа с ними завершена. К таким, например, относятся объекты `Scanner`, когда они не созданы от стандартных потоков ввода-вывода (как `System.in`). После завершения операций со `Scanner`, вне зависимости, прошли они успешно или нет, необходимо вызвать метод `.close()`. Как раз в этом случае и подойдёт использование `finally`.

Код в `finally` выполняется и если исключения не было, и после того, как оно было обработано в `catch`. То есть если в коде есть `finally`, то действие в нём выполнится почти всегда, кроме тех случаев, если в блоке `try` были вызваны методы, которые инициируют остановку JVM.

```
try {
    // действие, которое может вызвать ошибку
} catch (Exception exception) {
    // действие по обработке исключений
} finally {
    // действие, которое должно вызваться всегда
}
```

`finally` выполнится даже в тех случаях, если в любом из блоков `catch` произошло исключение или мы сгенерировали своё.

Блок `finally` можно использовать и при отсутствии блока `catch`. К примеру, в том случае, если вместо непроверяемого исключения будет брошено проверяемое, и оно будет отражено в сигнатуре метода при помощи `throws`. Тогда выполнение программы не будет прервано. Использование `finally` без `catch` может быть полезным, если нужно выполнить какие-то действия вне зависимости от того, произошло исключение или нет, но при этом его не требуется обрабатывать, так как оно может быть использовано дальше.

Если в `finally` произошло исключение, то в терминале вы увидите именно его стек-трейс, а не сообщение об ошибке, обработанной ранее в блоке `catch`. Чтобы избежать подобного — нужно добавить дополнительный обработчик `try...catch` для опасного кода внутри блока `finally`.

Также блок `finally` будет полезен, если необходимо выполнить какое-то завершающее действие в конце вне зависимости, случилась исключительная ситуация или нет. Однако используйте его внимательно — без `return` и предупреждая возможные сбои дополнительной обработкой!

В блоке `finally` может быть любой код, однако лучше не использовать в нём ключевое слово `return`. Сочетание `finally` и `return` приводит к тому, что нарушается логика программы.

Работа с файлами

Файлы в java

Библиотека `java.io.File`

Объект класса `File` нужен для управления информацией о файлах и директориях. У объекта типа `File` есть три варианта конструктора:

- `File(String)` — в строке абсолютный или относительный путь к файлу или каталогу, с которыми предполагается работа в коде;

- `File(File parent, String child)` — указать объект класса `File` (директория `parent`) и имя файла (`child`);
- `File(String parent, String child)` — указать путь к директории и имя файла.

> 💡 **Путь** (англ. *path*) — это набор символов, показывающий расположение файла или директории в файловой системе. Есть два вида путей:
> — **абсолютный (полный) путь** указывает на одно и то же место в файловой системе вне зависимости от текущей директории. Полный путь всегда начинается с корневого каталога;
> — **относительный (сокращённый) путь** указывает место относительно какой-либо отправной точки (другого файла, программы и так далее).

В классе `File` есть множество методов и свойств для работы с файлами и директориями. При необходимости вы можете обратиться к официальной документации и найти интересующее описание. Вот некоторые из них:

- `String getName()` позволяет узнать краткое имя файла или директории;
- `boolean isFile()` возвращает значение `true`, если по указанному пути находится файл;
- `boolean isDirectory()` возвращает значение `true`, если по указанному пути находится директория;
- `String[] list()` возвращает массив имён файлов и поддиректорий;
- `File[] listFiles()` возвращает массив объектов файлов и поддиректорий;
- `boolean mkdir()` создаёт новую директорию. При успешном создании возвращает значение `true`;
- `boolean renameTo(File dest)` переименовывает файл или директорию. В параметре указывается новое имя файла. Если переименовать не удалось, метод возвращает `false`;
- `boolean delete()` удаляет или файл, или пустую директорию по пути, который передан в конструктор. При успешном удалении возвращает `true`.

Также в классе `File` существует константа `separator`. С помощью неё вводятся разделительные знаки `/` или `\`.

Библиотека `java.nio.file.Path`

В 7-й версии Java создатели языка решили изменить работу с файлами и директориями. Вместо единого класса `java.io.File` появились три структуры:

- Класс `Paths`
- Интерфейс `Path` (документация)
- Класс `Files` (документация)

Класс `Paths` — небольшой класс с двумя статическими методами `get`, которые различаются только входными параметрами. Класс `Paths` создали исключительно для того, чтобы из переданной строки или `URI` получить объект типа `Path`:

- `Path get(String first, String... more)` преобразует строку пути (параметр `first`) или последовательность строк (параметр `more`), образующих при соединении строку пути, в `Path`;
- `Path get(URI uri)` преобразует заданный объект типа `URI`.

Интерфейс `Path` содержит имена директорий и файлов, которые составляют полный путь к файлу или каталогу. В нём также есть методы для добавления элементов пути, их извлечения и манипуляций с ними. Некоторые из методов `Path`:

- `Path getFileName()` возвращает имя файла из пути;
- `Path getParent()` возвращает «родительскую» директорию по отношению к текущему пути (ту, которая находится выше по дереву директорий);
- `Path getRoot()` возвращает «корневую» директорию (ту, которая находится на вершине дерева директорий).

Класс `Files` — это `final`-класс с `private`-конструктором. Он содержит только статические методы для выполнения различных действий. Его основные методы:

- `Path createFile(Path path, FileAttribute<?>... attrs)` создаёт новый пустой файл. Выбрасывает исключение, если файл уже существует. Параметры метода: `path` — путь к файлу, который нужно создать, `attrs` — необязательный список атрибутов файла (в нём можно указать правила доступа к файлу, добавить информацию о создателе и так далее).
- `Path createDirectory(Path dir, FileAttribute<?>... attrs)` создаёт новую директорию. Параметры метода: `dir` — директория, которую нужно создать, `attrs` — необязательный список атрибутов директории.
- `Path move(Path source, Path target, CopyOption. . . options)` перемещает файл. Параметры метода: `source` — путь к файлу, который нужно переместить, `target` — путь к файлу назначения, `options` — необязательные параметры, определяющие, как нужно делать перемещение.
- `void delete(Path path)` удаляет файл или директорию. Если удаляется директория, необходимо убедиться, что она пуста, иначе будет получено исключение

`DirectoryNotEmptyException`. Если удаляется файл, необходимо убедиться, что он существует, иначе будет получено исключение `NoSuchFileException`. Параметры метода: `path` — путь к файлу или директории, которые нужно удалить.

- `boolean deleteIfExists(Path path)` удаляет файл или директорию, если они существуют. Параметры метода: `path` — путь к файлу, который нужно удалить. Возвращаемое значение: `true` — если файл был удалён этим методом, `false` — если файл не может быть удалён, потому что не существует.

Метод `Path copy(Path source, Path target, CopyOption... options)` — то, чего не хватало в библиотеке File. Его параметры: `source` — путь к исходному файлу, `target` — путь к тому файлу, что будет создан в результате копирования (включая имя нового файла), `options` — необязательные параметры копирования. Существует три таких параметра:

- `REPLACE_EXISTING` указывает, что если в директории назначения уже есть такой файл, то нужно его заменить;
- `COPY_ATTRIBUTES` указывает, что нужно скопировать атрибуты оригинального файла в его копию;
- `ATOMIC_MOVE` указывает, что необходимо переместить файл. Это значит, что перемещение или выполнится целиком, или не выполнится вообще.

Обратите внимание: при копировании директории содержащиеся в ней файлы и каталоги копироваться не будут.

Поток

Поток (англ. *stream*) — это бесконечная последовательность данных. Поток подключён к источнику (англ. *source*) или получателю данных (англ. *destination*). По направлению потоки делятся следующим образом:

- **потоки ввода** (англ. *input streams*), из которых считываются данные;
- **потоки вывода** (англ. *output streams*), в которые записываются данные.

А по типу передаваемых данных так:

- **символьные потоки** (англ. *character streams*), которые содержат символы;
- **байтовые потоки** (англ. *byte streams*), которые содержат информацию в виде последовательности байтов.

Классы для работы с потоками

В Java все необходимые классы для работы с потоками ввода-вывода находятся в пакете `java.io`. Благодаря этим классам разработчику не нужно вникать в особенности низкоуровневой организации операционных систем.

Для каждого из типов потоков Java предлагает отдельный базовый абстрактный класс:

- `InputStream` представляет поток ввода для чтения байтов;
- `OutputStream` представляет поток вывода для записи байтов;
- `Reader` представляет поток ввода для чтения символов;
- `Writer` представляет поток вывода для записи символов.

`PrintStream` — стандартный выходной поток, который открыт во время выполнения программы и готов принимать выходные данные для вывода в терминал.

Потоки и файлы

Файлы — самые распространённые источники, или получатели данных в приложении. Для работы с файлами у каждого из четырёх абстрактных классов потоков есть своя реализация: `FileInputStream` и `FileOutputStream`; `FileReader` и `FileWriter`.

Выбор между байтовыми и символьными потоками зависит от того, с каким типом файла предстоит работать. Для бинарных файлов, таких как картинки, видео, pdf-файлы, нужен байтовый поток (`FileInputStream` для чтения и `FileOutputStream` для записи). Для текстовых файлов лучше использовать символьный поток (`FileReader` для чтения и `FileWriter` для записи), хотя можно применять и байтовые потоки.

Общая схема работы с потоками и файлами в Java выглядит так:

- Создаётся потоковый объект и ассоциируется с файлом на диске.
- Данные читаются из потока или записываются в поток.
- Поток закрывается.

Запись

Чтобы сделать запись в файл при помощи `FileWriter`, нужно сначала создать объект `FileWriter`. Далее с помощью метода `write()`, который есть у всех потоков вывода, можно добавить строки в новый файл. В конце необходимо закрыть поток методом `close()`.

💡 С конструктором `FileWriter(String fileName)` содержимое файла будет создаваться заново при каждом запуске программы. Чтобы добавить содержимое к уже существующему файлу, необходимо воспользоваться конструктором `FileWriter(String fileName, boolean append)` и передать в него значение `true` для флага `append` (англ. «присоединять»). Это специальный признак того, что новые данные будут записаны в конец файла.

Чтение

Чтобы прочитать текстовый файл, необходимо сначала создать объект `FileReader`, который подключается к файлу. `FileReader` считывает данные по одному символу за раз, пока не будет достигнут конец файла.

Метод `read()` возвращает значение типа `int`. А `int` содержит значение `char` прочитанного символа. Если метод `read()` возвращает `-1`, значит, в `FileReader` больше нет данных для чтения и его можно закрыть с помощью `close()`.

Буферизация

Буферизация — способ ввода и вывода данных, при котором для их временного хранения используется область памяти — **буфер**. Буфером может быть обычный массив. Данные по очереди попадают в него, накапливаются, а затем обрабатываются вместе. Если добавить буферизацию, `FileReader` будет обращаться за символами не к файлу, а к буферу. Это увеличит производительность программы. Буферизация используется не только для чтения файлов, но и для записи. `BufferedReader` — подкласс `Reader`, а значит, он может использовать все те методы для чтения из потока, которые определены в классе `Reader`.

```
BufferedReader(Reader in)
BufferedReader(Reader in, int sz)
```

Обратите внимание, что в конструкторе `BufferedReader(Reader in, int sz)`, кроме потока ввода, из которого производится чтение, нужно также указать размер буфера, в который будут считываться символы. Если не передать размер буфера в конструктор явно, программа использует значение по умолчанию — 8192 символа.

`try-with-resources`

Операционная система контролирует совместный доступ разных программ к файлам. Если поток больше не нужен, то его необходимо закрыть методом `close()` вот так:

```
FileOutputStream fos = new FileOutputStream("file.txt");
// что-то делаем с потоком
```

```
fos.close();
```

Важно помнить, что исключение может возникнуть до вызова метода `close()`. Тогда поток не будет закрыт. Можно вызвать `close()` в блоке `finally`. Однако это будет малоэффективно, если ошибка возникнет при создании потока.

До 7-й версии Java правильное закрытие было громоздким и запутанным. Поэтому появилась конструкция **try-with-resources** (англ. «try с ресурсами»), которая позволяет закрывать один или несколько ресурсов без использования блока `finally`.

Под ресурсом понимается любой класс, наследуемый от интерфейсов `Closeable` или `AutoCloseable`. В этих интерфейсах объявлен метод `close()`, который необходимо реализовать.

Метод `close()` будет вызван автоматически, когда программа выйдет из блока `try-with-resources`. В блоке `try-with-resources` можно объявить несколько ресурсов. Тогда их необходимо разделить точкой с запятой. При этом ресурсы, которые были определены первыми, будут закрыты последними.

Обратите внимание: у `try-with-resources`, так же как и у обычного `try`, тоже могут быть блоки `catch` и `finally`. Чтобы преобразовать стандартный `try` в `try-with-resources`, достаточно объявить необходимые ресурсы в круглых скобках после ключевого слова `try`, а остальное сделает Java.

Кодировки и Java

Если не указать тип кодировки, программа не сможет правильно отобразить текст и выведет на экран бессмысленную последовательность символов.

Для представления кодировок в Java существует специальный класс `Charset`, а также класс `StandardCharsets` с константами для стандартных наборов символов.

В классе `Charset` есть метод `static Charset forName(String charsetName)`, который возвращает объект кодировки по имени. В качестве параметра `charsetName` может быть как стандартное имя кодировки, так и её псевдоним. То есть значения `utf8`, `UTF-8`, `utf-8` будут распознаны одинаково.

ASCII, ISO 8859 и кодировки Windows определяет 128 символов. Каждый из них сопоставляется с числовым значением от 0 до 127. Таблица начинается с 32 невидимых управляющих символов и заканчивается символом DEL, тоже управляющим. Символы в диапазоне от 32 до 126 относятся к видимым — это пробел, знаки препинания, латинские буквы и цифры.

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	LF	11	VT	12	FF	13	CR	14	SO	15	SI
16	DLE	17	DCI	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	“	35	#	36	$	37	%	38	&	39	‘
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

В отличии от ASCII набор символов Юникода разбит на 17 плоскостей, содержащих по 65 536 числовых значений. Таким образом, максимально возможное число символов равно 1 114 112.

Юникод реализован несколькими способами, но самый распространённый — **UTF-8**. (от англ. *Unicode Transformation Format, 8-bit,* «формат преобразования Юникода, 8-бит»). Это относительно новый стандарт кодирования, который позволяет закодировать не только английский алфавит, но и китайские иероглифы, арабскую вязь, дополнительные типографские символы и даже эмодзи.

В программах на Java часто встречаются места, где кодировка должна быть указана явно. Если этого не сделать, будет использована **кодировка по умолчанию** (англ. *default charset*). Она определяется во время запуска виртуальной машины и сохраняется в свойстве `file.encoding`. Значение зависит от выбранного языка и кодировки самой операционной системы.

Кодировка по умолчанию — глобальный параметр. Нельзя установить для одних классов или функций одну кодировку, а для других — другую. Кроме того, её нельзя изменить в процессе выполнения программы. Но главное, разработчик может использовать функцию, которая будет работать по-разному в разном окружении, и не заметить этого. Определить кодировку по умолчанию можно с помощью метода `defaultCharset()` класса `Charset`.

Проблему кодировки по умолчанию решает запуск Java с параметром `-Dfile.encoding=UTF-8`.

```
java -Dfile.encoding=UTF-8 Practicum
```

Для основных классов потоков предусмотрен конструктор, в который можно передать нужную кодировку. Например, `InputStreamReader(InputStream in, Charset cs)`, `PrintStream(boolean autoFlush, OutputStream out, Charset charset)` и другие.

При работе с массивом байтов тоже следует указывать желаемую кодировку, иначе можно столкнуться с неприятностями из-за кодировки по умолчанию. `byte[] getBytes(Charset charset)` и `String(byte bytes[], String charsetName)`.

```
// преобразование из строки в массив байтов
String s = "Обычная строка.";
byte[] buffer = s.getBytes(StandardCharsets.UTF_8);

// преобразование из массива байтов в строку
byte[] buffer = new byte[1000];
String s = new String(buffer, StandardCharsets.UTF_8);
```