Для представления файлов в Java существуют два класса – ***java.io.File*** (старый) и ***java.nio.file.Path*** (начиная с 7 версии).

**Абсолютный путь** – это полный путь, начиная с корневого каталога.

**Относительный путь** – это путь относительно какой-либо отправной точки (например, текущей).

Старый класс ***File*** использовался для управления информацией о файлах и директориях. У него есть два конструктора *File(File parent, String child)* и *File(String parent, String child)*.

Метод ***getName()*** возвращает строку с именем файла или директории.

Метод ***isFile()*** возвращает булево значение определяющее наличие файла.

Метод ***isDirectory()*** возвращает булево значение определяющее наличие директории.

Метод ***list()*** возвращает массив имен файлов и поддиректорий *String[]*.

Метод ***listFiles()*** возвращает массив объектов файлов и поддиректорий *File[]*.

Метод ***mkdir()*** создает новую директорию и возвращает *true* при успешном создании.

Метод ***renameTo(File)*** переименовывает файл или директорию на имя в параметре. Возвращает *false* при неудаче.

Метод ***delete()*** удаляет файл или пустую директорию и возвращает *true* при успехе.

В разных системах используется разные разделительные знаки, поэтому в классе *File* есть константа ***separator***, которая автоматически адаптируется под операционную систему.

*File dir = new File(HOME + File.separator + "NewDir");*

У класса *File* отсутствуют методы копирования и перемещения файлов, и слишком много методов возвращают булево значение, поэтому его переработали в 7й версии *Java*. В новой версии появилось три структуры:

1. Класс ***Paths*** – служебный класс с двумя статическими методами get для получения объекта типа Path. Методы отличаются только параметрами.

Метод ***get(String first, String… more)*** преобразовывает строку пути или последовательность строк образующих при соединении строку пути в *Path*.

Метод ***get(URI uri)*** преобразовывает символьную строку на ресурс (*file:///C:/books/dune.epub*) в объект *Path*.

1. Интерфейс ***Path*** – доработанная версия *File*, у которого есть множество реализаций для конкретных файловых систем. Он содержит методы для добавления элементов пути, их извлечения и манипуляций с ними.

Метод ***Path getFileName()*** возвращает имя файла из пути.

Метод ***Path getParent()*** возвращает родительскую директорию по отношению к текущему пути.

Метод ***Path getRoot()*** возвращает корневую директорию.

1. Класс ***Files***, состоящий из множества статических методов для работы с файлами.

Метод ***Path createFile(Path path, FileAttribute<?>… attrs)*** создает новый пустой файл. Если файл существует – выбрасывает исключение. Необязательный список атрибутов *attrs* может содержать правила доступа к файлу, информацию о создателе и т.д.

Метод ***Path createDirectory(Path dir, FileAttribute<?>… attrs)*** создает новую директорию.

Метод ***Path move(Path source, Path target, CopyOption… options)*** перемещает файл.

Метод ***void delete(Path path)*** удаляет файл или пустую директорию. Если директория не пуста или файла не существует – выбросит исключение.

Метод ***boolean deleteIfExists(Path path)*** удаляет файл или директорию, если они существуют.

Метод ***Path copy(Path source, Path target, CopyOption… options)*** копирует указанный объект. При копировании директории содержащиеся в ней файлы и каталоги не будут скопированы. В *options* могут быть до трех параметров:

* *REPLACE\_EXISTING* – заменить файл, если он уже существует в директории назначения.
* *COPY\_ATTRIBUTES* – скопировать атрибуты оригинального файла в его копию.
* *ATOMIC\_MOVE* – переместить файл **атомарно** (либо все, либо ничего).

**Поток** – это бесконечная последовательность данных. Поток подключен к источнику *source*и получателю *destination*. В *java* определены **байтовые** (байты, они же символы в коде *ASCII*) и **символьные** потоки (символы в коде *Unicode*). Символьные потоки являются адаптированными байтовыми потоками.

Все необходимые классы для работы с потоками ввода-вывода находятся в пакете *java.io*. Для каждого из типов поток *Java* предлагает отдельный базовый абстрактный класс.



Помимо этих потоков, в программы автоматически импортируется пакет *java.lang* с классом *System*, в котором определены переменные ***in***, ***out*** и ***err***, представляющие стандартные байтовые потоки ввода-вывода. Они являются экземплярами *InputStream* (для первого) и *PrintStream* (для остальных) и поддерживают их методы.

Для работы с файлами у каждого из четырех абстрактных классов потоков есть своя реализация:

* *FileInputStream* и *FileOutputStream* лучше использовать для бинарных файлов (картинки, видео, pdf).
* *FileReader* и *FileWriter* лучше использовать для текстовых файлов, хотя можно применять и байтовый поток.

Общая схема работы с потоками и файлами:

1. Создается потоковый объект и ассоциируется с фалом на диске
2. Данные читаются из потока или записываются в поток
3. Поток закрывается.

При возникновении ошибок чтения/записи потоком выдается исключение типа ***IOException***.

У класса *FileWritter* существует несколько конструкторов. Первый ***FileWriter(String)*** – переписывает файл каждый раз заново, второй ***FileWriter(String, boolean)*** – при передаче *true* добавляет данные в конец файла.

*public static void main(String[] args) throws IOException {*

*Writer fileWriter = new FileWriter("filewriter.txt"); // откроем поток*

*fileWriter.write("новая запись в новый файл\n"); // добавим строку*

*fileWriter.close(); } // закроем поток*

Класс *FileReader* считывает данные по одному символу пока не достигнет конца файла. Метод int read() возвращает значение char прочитанного символа. Если он вернет -1, значит файл закончился.

*public static void main(String[] args) throws IOException {*

*Reader fileReader = new FileReader("filereader.txt");*

*int data = fileReader.read();*

*while (data != -1) {*

*System.out.print((char) data);*

*data = fileReader.read(); }*

*fileReader.close(); }*

В примерах выше класс *FileReader* обращался за каждым символом напрямую к файлу. Этот процесс можно ускорить, применив **буферизацию** – способ ввода/вывода данных, при котором для их временного хранения используется область памяти (**буфер**). При создании буфера можно указать его размер. Например, в классе ***BufferedReader*** по умолчанию используется 8192 символа.

*public static void main(String[] args) throws IOException {*

*FileReader reader = new FileReader("bufferedreader.txt");*

*BufferedReader br = new BufferedReader(reader); // буферизация*

*while (br.ready()) {*

*String line = br.readLine();*

*System.out.println(line); }*

*br.close(); }*

Начиная с JDK 7, появился автоматизированный процесс освобождения ресурсов конструкцией ***try-with-resources***. Под ресурсами понимается любой класс, наследуемый от интерфейсов ***Closeable*** или ***AutoCloseable***. При необходимости можно закрыть доступ явным образом через метод *close()*. В блоке *try-with-resources* можно объявить несколько ресурсов, разделенных точкой с запятой, однако ресурсы, открытые первыми, будут закрыты последними.

*public class Practicum {*

*public static void main(String[] args) {*

*try (Resource1 resource1 = new Resource1(); Resource2 resource2 = new Resource2()) {*

*System.out.println("внутри блока try"); } } }*

*class Resource1 implements AutoCloseable { // создаем свой класс*

*@Override*

*public void close() { // и реализуем интерфейс*

*System.out.println("метод close() для Resource1"); } }*

**Кодировка** – набор числовых значений, в котором каждое значение соответствуют своему символу.

Кодировка ***ASCII*** (***American Standart Code for Information Interchange***) определяет 128 символа от 0 до 127. Таблица начинается с первых 32 невидимых управляющих символов и заканчивается невидимым управляющим символом *DEL*. Между ними располагаются видимые пробел, знаки препинания, латинские буквы и цифры.

Кодировка ***Unicode*** разбит на 17 плоскостей, содержащих по 65536 числовых значений (всего 1 114 112). Она включает все современные алфавиты, цифры, знаки препинания, математические, денежные, музыкальные символы и символы мертвых языков. Самый распространенный вариант – 8-битный ***UTF-8***.

Для представления кодировок в *java* существует специальный класс ***Charset*** и пакет *java.nio.charset.StandardCharsets*. Задать кодировку можно либо с помощью класса ***StandardCharset*** с константами для стандартного набора символов, либо через статический метод ***Charset*** ***forName(String charsetName)***. Первый способ предпочтительнее.

Если не задать кодировку явно, то будет использована кодировка по умолчанию (на основании выбранного языка и операционной системы). Определить кодировку по умолчанию можно так:

*System.out.println(Charset.defaultCharset().displayName());*

Проблему кодировки по умолчанию решает запуск *Java* с параметром. Например:

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

Для основных классов поток предусмотрены конструкторы, в которые можно передать кодировку. Например, ***InputStreamReader(InputStream in, Charset cs)***, ***PrintStream(boolean autoFlush, OutputStream out, Charset charset)*** и другие.

Многие методы могут работать только с массивом байтов, поэтому есть соответствующие методы ***getBytes()***, преобразующий строку в байтовый массив, и ***String(byte bytes[])***, создающий строку из байтового массива. В них тоже можно указывать кодировку в качестве параметра.