Лекция 12.
Работа со строками.
Работа с файлами

Строки в Java

- Строки являются объектами
- Строки являются **неизменяемыми объектами**, т.е. объект строки нельзя изменить после его создания
- Поэтому все функции, которые работают со строками, возвращают новые строки, а старые – остаются без изменения
- Строки, как и все объекты, нужно сравнивать через **equals**()

Документация по строкам

- http://docs.oracle.com/javase/8/docs/api/java/lang/String.html

- Там перечислены все функции строк и их описание

Функции для работы со строками

- int compareTo(String s)
- Сравнивает строки лексикографически:
  - возвращает 0, если строки равны (по **equals**),
  - положительное число, если данная строка больше переданной
  - отрицательное число – в противном случае
-
```
String a = “123”;
if (a.compareTo(“344”) > 0) {
    // код
}
```

Лексикографическое сравнение строк

- Сравниваем коды первых символов строк. Если один из кодов больше, то это строка больше
- Если коды равны, переходим к проверке следующих символов и т.д.
- Если при этом одна из строк кончилась, то она считается меньше

- Пример верных высказываний:
- “abc” меньше “b”, “ab” > “a”

Функции для работы со строками

- int compareToIgnoreCase(String s)
- То же самое, только без различия регистра

Contains

- boolean contains(String s)
- Проверяет, входит ли переданная строка в данную строку

- Пример:
- String s = “1000 dollars”;
- if (s.contains(“1000”)) {
  System.out.println(“Есть 1000”);
  }

EndsWith, startsWith

- boolean endsWith(String s)
- Проверяет, заканчивается ли текущая строка на переданную строку

- boolean startsWith(String s)
- Проверяет, начинается ли текущая строка на переданную строку

IndexOf

- int indexOf(String s)
- Выдает первый индекс, начиная с которого в текущей строке находится переданная строка. Если переданной строки нет в строке, то выдается -1

- Пример:
- String s = “Of 1004 1004 1004”;
- int index = s.indexOf(“1004”); // 3

IndexOf

- int indexOf(String s, int startIndex)
- Аналогично, только ищет, начиная с переданного индекса startIndex

- Пример:

```
String s = “Of 1004 1004 1004”;
int index = s.indexOf(“1004”, 4);   // 8
```

lastIndexOf

- int lastIndexOf(String s)
- Выдает последний индекс, начиная с которого в текущей строке находится переданная строка. Если переданной строки нет в строке, то выдается -1

- Пример:

```
String s = “Of 1004 1004 1004”;
int index = s.lastIndexOf(“1004”);   // 13
```

isEmpty

- boolean isEmpty()

- Проверяет, что строка равна пустой строке (строке длины 0)

Replace

- String replace(String toSearch, String replacement)

- Заменяет все вхождения первой переданной строки на вторую переданную строку

- ```
  String s = “1004 1004 1004”;
  s = s.replace(“1004”, “1005”);
  // 1005 1005 1005
  ```
  ```

Split

- String[] split(String s)
- Разбивает строку на массив подстрок по указанной строке-разделителю

- Пример:

```
String numbersLine = “1, 2, 3”;
String[] numberStrings = numbersLine.split(“, ”);
// массив из элементов “1”, “2” и “3”
```

toLowerCase, toUpperCase

- String toLowerCase()
- Переводит новый объект строки, который содержит текущую строку, но в нижнем регистре

- String toUpperCase()
- Аналогично, только в верхнем регистре

trim

- String trim()
- Возвращает новый объект строки, который содержит текущую строку, но в которой обрезаны пробельные символы в начале и конце строки

- Пример:
- String s = “ 123\t ”;
  s = s.trim(); // “123”

Substring

- String substring(int startIndex, int endIndex)
- Возвращает подстроку, начиная с начального индекса startIndex и заканчивая конечным endIndex, не включая символ по конечному индексу
- Можно указать только начальный индекс, тогда подстрока возьмется до конца строки
- Пример:

```
String s = “123 456”;
s = s.substring(4, 7);     // “456”
```

- Аналогично: s = s.substring(4); // “456”

Преобразование строк в числа

- Integer.parseInt(String s)
- Double.parseDouble(String s)

- Пример:
- int a = Integer.parseInt(“345”); // 345
- double b = Double.parseDouble(“3.2”); // 3.2

StringBuilder

- Класс StringBuilder используется для формирования больших строк
- StringBuilder sb = new StringBuilder();

```
sb.append(“Номер квартиры = ”)
  .append(flatNumber)
  .append(“, номер подъезда = ”)
  .append(entranceNumber);

String result = sb.toString();
// получение результирующей строки
```

Вызовы append можно составлять в цепочки

Это потому что append делает в конце return this;

StringBuilder

- Важные методы:
  - append(data) – вставка в конец
  - delete(int startIndex, int endIndex) – удаление символов от начального индекса до конечного, не включая конечный индекс
  - toString() – преобразование в строку
  - length() – получение длины строки
  - insert(int index, data) – вставка в середину
  - setCharAt(int index, char c)
  - deleteCharAt(int index)

Задача «StringBuilder»

- Создать строку из чисел от 1 до 100 через запятую при помощи StringBuilder
- Распечатать строку в консоль

- Не называйте свой класс StringBuilder, чтобы не было конфликтов имен со стандартным классом

Задача «URL»

- Написать функцию, которая вычленяет из URL адреса имя сервера. Имеется в виду следующее. Для строки вида http://SomeServerName/abcd/dfdf.htm?dfdf=dfdf вычленить SomeServerName
- Строка может начинаться не обязательно с http, но и с https или чего-то другого. Но :// есть всегда
- Учесть случай, когда после :// больше нет слэша:
- http://SomeServerName
- Использовать indexOf и substring

Задача «Разбиение строки»

- Разбить строку “1, 2, 3, 4, 5” и получить массив из этих чисел и найти их сумму

- Использовать split и Integer.parseInt

Работа с файлами

Пример файла

- Первое число n – целое, означает количество чисел
- Далее идёт n вещественных чисел

- Пример:

  3 1,3 4,4 5,5

- Хотим прочитать файл и положить вещественные числа в массив

Чтение файлов

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        // создаем сканнер от FileInputStream(имя файла)
        Scanner scanner = new Scanner(new FileInputStream("input.txt"));

        // дальше работаем со сканнером как обычно
        int count = scanner.nextInt();
        double[] a = new double[count];
        for (int i = 0; i < count; ++i) {
            a[i] = scanner.nextDouble();
        }

        // когда мы все прочитали, сканнер нужно закрыть
        scanner.close();
    }
}
```

Чтение файла отличается от чтения с консоли только параметром конструктора сканнера

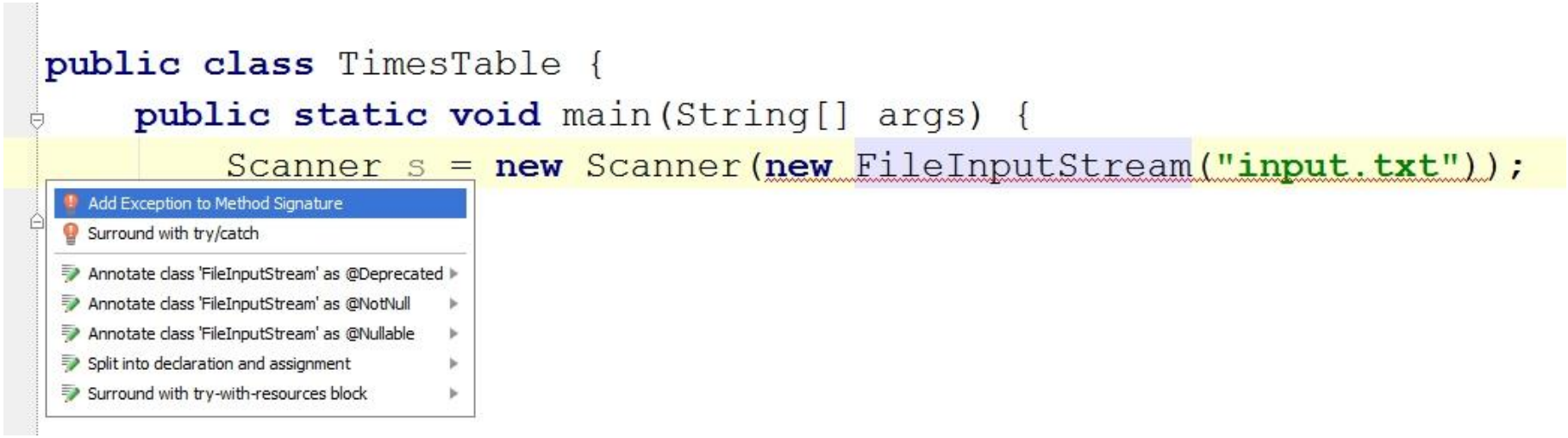
И тем, что сканнер нужно закрывать после работы, чтобы освободить ресурсы

Ошибка при создании потока чтения

- При попытке создать Scanner, который будет читать из файла, Java выделит создание объекта FileInputStream красным
- Это произойдет потому что открытие файла может завершиться неуспешно – например, файла не существует. Тогда при вызове конструктора произойдет ошибка
- Чтобы избавиться от ошибки, есть 2 варианта:
  - Указать при объявлении функции, что она может завершиться ошибкой
  - Написать специальный код, который будет обрабатывать ошибку

Ошибка при создании потока чтения

- Мы пока что будем указывать, что функция может завершиться с ошибкой:
  1. Наводим курсор на подчеркиваемый код
  2. Наживаем Alt+Enter
  3. Выбираем “Add Exception to Method signature”



Ошибка при создании потока чтения

- После этого в сигнатуру метода добавится строка:

```
public class TimesTable {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner s = new Scanner(new FileInputStream("input.txt"));
    }
}
```

- Ее смысл в том, что в функции может произойти такая ошибка: FileNotFoundException

Частые ошибки при запуске примера

- Файл находится не в той папке:
  - Нужно создавать файл именно в корневой папке проекта, а не внутри **src**
- Неверное имя файла:
  - В Windows по умолчанию скрываются расширения файлов, лучше настроить Проводник так, чтобы расширения показывались
  - Убедиться, что имя файла вместе с расширением совпадает с тем, что написано в коде
- Неверные данные в файле:
  - Должна быть запятая для дробной части вещ-х чисел
  - Убедиться, что содержимое файла соответствует логике задания

Правильная работа с файлами

Закрытие потока

```
public static void main(String[] args) throws IOException {
    try (Scanner scanner = new Scanner(
            new FileInputStream("input.txt"))) {

        // работаем со сканнером
        int x = scanner.nextInt();
    }
}
```

- После того, как работа со **scanner**'ом завершена, его обязательно нужно закрывать, вызвав метод **close**()
- Но лучше всего использовать конструкцию try, которая закрывает ресурсы при завершении блока
- Для потока System.in делать это не нужно

Правильное чтение файлов

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        // создаем сканнер от FileInputStream(имя файла)
        try (Scanner scanner = new Scanner(new FileInputStream("input.txt"))) {
            // дальше работаем со сканнером как обычно
            int count = scanner.nextInt();
            double[] a = new double[count];
            for (int i = 0; i < count; ++i) {
                a[i] = scanner.nextDouble();
            }

            // теперь close не нужен – он вызовется сам
        }
    }
}
```

Зачем нужен try?

- Допустим, у нас такой код:
- ```
  Scanner scanner = new Scanner(new FileInputStream("input.txt"));
  // произвольный код
  scanner.close();
  ```

- Сейчас если код в середине упадет с ошибкой или в нем, например, будет return или другая конструкция перехода, то **close** может не вызваться
- Тогда, если наша программа не завершится, то файл останется незакрытым
- И тогда другие программы не смогут с ним нормально работать, пока программа не завершится
  ```

Зачем нужен try?

- Конструкция try гарантирует, что **close** будет вызван в любом случае:
  - если блок try просто успешно завершится
  - если внутри кода в try будет ошибка
  - если внутри кода в try будет вызван return или другая конструкция для перехода
- Т.е. файл в любом случае закроется, и это правильно

Виды путей

Виды путей

- **Абсолютный путь** – это полный путь (в Windows - с полным указанием буквы диска и т.д.):
  - F:\Users\Pavel\IdeaProjects\Test\folder\input.txt
- **Относительный путь** - это путь относительно некоторой папки (в IDEA – это относительно корневой папки проекта):
  - folder\input.txt
- Этот путь указывает туда же, что и абсолютный путь в предыдущем примере при запуске из проекта, который находится здесь:
  - F:\Users\Pavel\IdeaProjects\Test

Путь к файлу

- **Относительный путь**

  input.txt

- Такой относительный путь означает, что файл **input.txt** лежит в корневой папке проекта

Специальные символы . и ..

- В относительных путях могут использоваться специальные символы . и ..
- Одна точка означает текущую папку
  Т.е. эквивалентно:
  - input.txt
  - ./input.txt
- Две точки означают родительскую папку
  - ../input.txt // находится в родительской папке

Еще про чтение файлов

Создание Scanner’а с кодировкой

- ```
  Scanner scanner =
      new Scanner(new FileInputStream("input.txt"), “windows-1251”);
  ```

- Чтобы задать кодировку сканнеру, ему в конструктор можно передать второй параметр - название кодировки
- Кодировка windows-1251 – это стандартная кодировка txt файлов в Windows
  ```

Методы hasNextXXX

- Класс Scanner имеет методы **hasNextDouble**(), **hasNextInt**(), **hasNextLine**()
- Метод **hasNextDouble**() проверяет, что следующая часть потока чтения является вещественным числом, и возвращает соответствующее boolean значение
- При этом, если поток закончился (всё прочитали), то тоже вернется false

Методы hasNextXXX

- **hasNextInt**(), **hasNextLine**() работают аналогично, только **hasNextInt** проверяет наличие целого числа в потоке, а **hasNextLine** – строки

- Есть метод **hasNext**(), который возвращает true, если в потоке есть еще что-нибудь
- Этот метод подходит, чтобы проверить, что файл кончился

Пример hasNextDouble

- Можно читать данные из сканнера, пока они не закончатся:

- ```
  while (s.hasNextDouble()) {
      // работаем с прочитанным числом
      double d = s.nextDouble();
      System.out.println(d);
  }
  ```
  ```

Чтение неизвестного количества данных

- Мы пока что изучили только массивы, а у массива фиксированная длина
- Если массив переполнился, то нужно создать новый массив, например, вдвое большей длины, и перекопировать данные туда. А дальше работать с этим новом массивом
- Если он тоже переполнится, то опять создать новый массив и т.д., пока все не будет прочитано

- Существуют классы, которые уже имеют внутри себя такую логику – это списки, они будут рассмотрены на курсе ООП

Работа с большими файлами

- Гипотетически файлы могут быть очень большими, и не влезать в оперативную память
- Поэтому лучше не считывать все содержимое файла в массив или в единую строку
- Лучше обрабатывать файлы построчно

- Чтение файла целиком допускается только если файл небольшой
- Во всех задачах с курса предполагается, что файлы могут быть большими

Что если формат файла более сложный?

- По сути файл придется разбирать вручную, есть 2 подхода, которые при желании можно комбинировать:
  - Читать из файла строки. Разбирать эти строки вручную через функции работы со строками (например, **split**, **substring** и др.)
  - Использовать **регулярные выражения**

- В любом случае вы должны знать формат файла, иначе прочитать его не удастся

Регулярные выражения

- **Регулярные выражения** – это специальный язык для поиска подстрок в тексте
- Регулярные выражения позволяют найти подстроки, соответствующие некоторому **шаблону**, а потом с этими подстроками можно работать – получать, заменять, удалять их и т.д.

Регулярные выражения – пример задачи

- Допустим, мы хотим найти все даты в этом тексте:
- Матч Египет-Уругвай прошел **15.06.2018**, а матч Португалия-Испания прошел **16.06.2018**.

- Чтобы найти даты, нам понадобится такое регулярное выражение: **\d{2}\.\d{2}\.\d{4}**
- **\d** - означает цифра (т.е. символ 0-9)
- **\.** – означает символ «точка». Здесь нужно экранирование, т.к. у точки в регулярных выражениях особый смысл
- **{n}** – это означает, что последовательность должна встречаться **n** раз

Регулярные выражения – пример задачи

- **\d{2}\.\d{2}\.\d{4}**
- В итоге получаем, что сначала идет 2 цифра, потом точка, потом еще 2 цифры, потом точка, потом 4 цифры
- Все подстроки такого вида могут быть найдены

- Во всех языках программирования есть некоторые функции или классы для работы с регулярными выражениями
- Кстати, в Java функция **split** принимает именно регулярное выражение, а не просто строку

Запись в файл

Запись в файл

- ```
  PrintWriter writer = new PrintWriter("output.txt");
  writer.println("OK!");
  writer.close();
  ```

- Класс PrintWriter имеет те же методы, что System.out, т.е. можно использовать **print**, **println**
- Все это будет записываться в файл
- Файл с указанным именем будет создан если его нет, либо перезаписан, если файл уже существует
- Как и при чтении, после окончания работы, **writer** нужно закрыть
  ```

Правильное закрытие потока

- ```
  public static void main(String[] args) throws IOException {
      try (PrintWriter writer = new PrintWriter("output.txt")) {
          // что-то пишем во writer
          writer.println(“OK”);
      }
  }
  ```
- После того, как работа с **writer’**ом завершена, его обязательно нужно закрывать, вызвав метод **close**()
- Но лучше всего использовать конструкцию try, которая закрывает ресурсы при завершении блока
  ```

Несколько потоков в try

- В try можно создавать несколько потоков, которые нужно будет автоматически закрыть
- Они указываются внутри круглых скобок try через точку с запятой
- ```
  public static void main(String[] args) throws IOException {
      try (Scanner scanner = new Scanner(new FileInputStream("input.txt"));
           PrintWriter writer = new PrintWriter("output.txt")) {
          while (scanner.hasNextLine()) {
              writer.println(scanner.nextLine());
          }
      }
  }
  ```

Копирование текстового файла, для примера
  ```

Работа с файлами в целом

Классы и методы для работы с файлами

- Мы рассматривали классы для работы с содержимым текстовых файлов
- Но часто бывает нужно работать с самими файлами (например, копировать, перемещать, удалять файлы или папки)
- Для работы с файлами можно использовать класс File, см. документацию:
- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html

Пример удаления файла

```
// создаем объект File, он создастся, даже если такого
// файла нет. Но при этом на диске файл не создается
File file = new File("out.png");

// поэтому мы проверяем, что файл действительно есть
if (file.exists()) {
    // метод delete пытается удалить файл
    // и выдает boolean – удалось удалить или нет
    if (file.delete()) {
        System.out.println("Файл успешно удален");
    } else {
        System.out.println("Не удалось удалить файл");
    }
}
```

Домашняя работа

Задача «Перевод файла в верх.регистр»

- Написать программу, которая читает строки файла, переводит их в верхний регистр и записывает результат во второй файл

Задача на дом «Число вхождений»

- Прочитать текст из файла, и написать функцию, которая считает количество вхождений некоторой строки в этот текст без учета регистра символов

- Использовать цикл и **indexOf**, который принимает начальный индекс, с которого искать