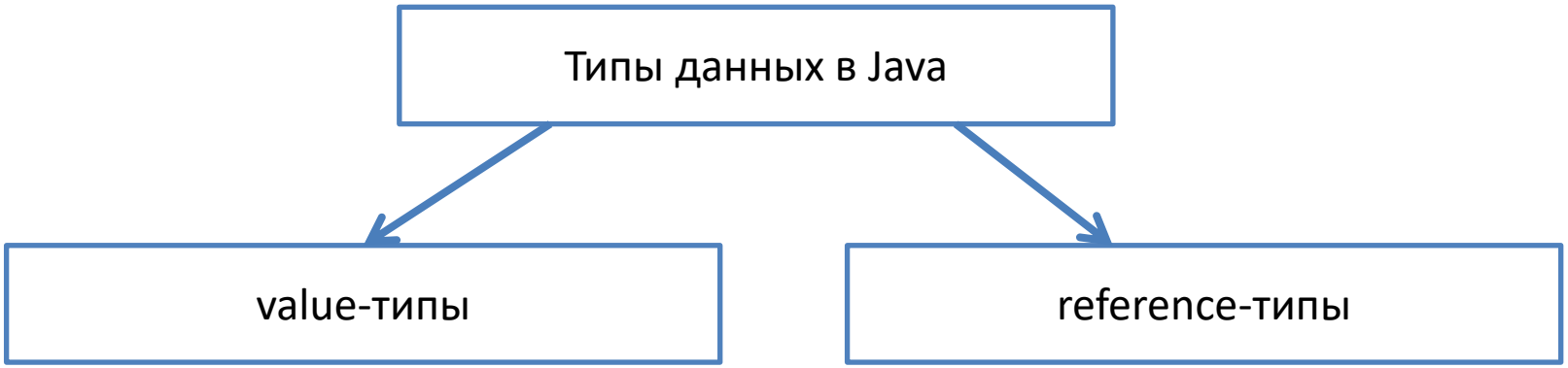
Лекция 2.
Value и reference-типы.
Статический инициализатор
Enum’ы.
Перегрузка методов

Типы данных в Java

- Все типы в Java можно разделить на две категории: **value-типы (типы значений)** и **reference-типы (ссылочные типы)**
- Типы из данных категорий ведут себя по-разному
- К value-типам относятся, например числа, а к reference-типам относятся строки

Типы данных в Java

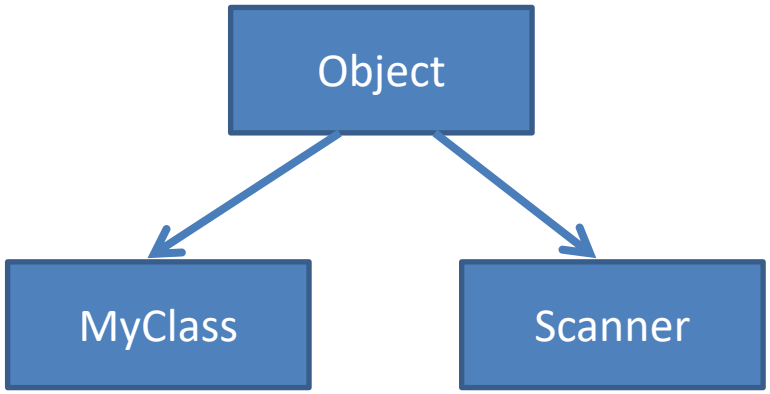


Числовые целые:
byte, short, int, long

Вещественные:
float, double

Логический: boolean

Символьный: char



Все классы наследуются от класса Object

Value-типы

- Их 8 штук
- Все числовые типы:
  byte, short, int, long; float, double
- Логический тип boolean
- Символьный тип char (рассмотрим позже)

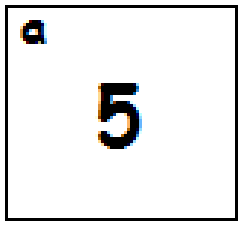
- В Java эти типы называют **примитивными**

Value-типы

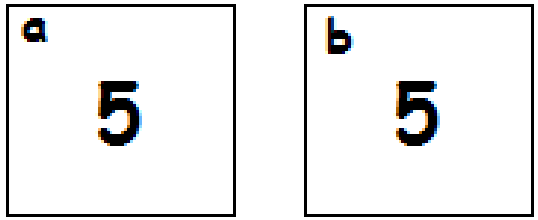
- Переменные value-типов хранят само значение типа
- При присваивании происходит копирование значения
- При передаче аргументов в функции, происходит копирование аргумента

Как работают value-типы

- int a = 5;



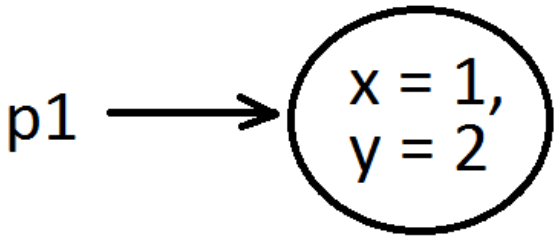
- int b = a;



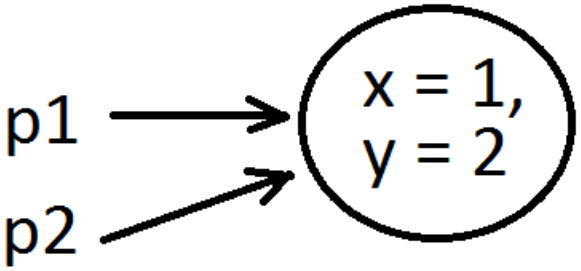
- Если изменить a или b, то это не повлияет на другую переменную

Reference-типы

- Переменные хранят не само значение, а **ссылку** на него (по сути – адрес в памяти)
- `Point p1 = new Point(1, 2);`



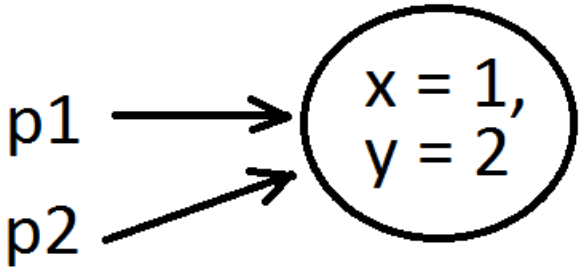
- При присваивании происходит копирование ссылки:
- `Point p2 = p1;`



- В Java все классы являются reference-типами

Reference-типы

- Если изменим объект,
  то все ссылки будут
  указывать на измененный объект



- p2.setX(3);
  System.out.println(p1.getX()); // 3
- При передаче объекта в функцию, происходит копирование ссылки на него
- Зачем нужны ссылки? Чтобы более эффективно работать с памятью. Объекты часто являются большими, и копировать их очень затратно по времени и памяти

Проверка объектов на равенство

- Для объектов нельзя использовать проверку через == и !=
- Для объектов оператор == проверяет, что ссылки указывают на один и тот же объект в памяти или нет
- Аналогично != проверяет, что ссылки указывают на разные объекты
- Чтобы сравнить содержимое объектов, нужно использовать метод equals
- boolean x = o1.equals(o2)

Значение null

- Переменные ссылочных типов могут принимать специальное значение null
- Пример: String s = null;
- Оно означает пустую ссылку, то есть адрес, который никуда не указывает
- Если вызвать функцию для переменной, которая имеет значение null, то произойдет ошибка NullPointerException

Для чего полезен null?

- Значение null может быть полезно, если мы хотим показать, что функция отработала, но получить результат не удалось
- Например, мы написали функцию, которая ищет строку нужной длины среди некоторого набора строк
- Но такой строки не оказалось
- В этом случае функция может вернуть null, а вызывающий код проверить, что результат равен null и, например, напечатать сообщение, что ничего не найдено

Для чего полезен null?

```
public static String findString(int length) {
    // код, который делает return, если нашел строку

    // в конце делается return null если
    // ничего не найдено
    return null;
}

public static void main(String[] args) {
    if (findString(4) == null) {
        System.out.println("Ничего не найдено");
    }
}
```

Перегрузка методов

- В одном классе можно создавать методы с одинаковыми именами, но разной **сигнатурой**
- Пример:
- ```
  public class Summator {
      public double sum(double a, double b) {
          return a + b;
      }

      public int sum(int a, int b) {
          return a + b;
      }
  }
  ```

Создание в классе разных методов с одинаковыми именами называется **перегрузкой метода (overloading)**
  ```

Сигнатура метода для перегрузки

- В **сигнатуру** для перегрузки входят: название метода, количество, типы и порядок аргументов
  ```
  int f(int a, double b)
  int f(double a, int b)   // имеют разные сигнатуры
  double f(double a)
  ```
- В классе нельзя определить два метода с одинаковой сигнатурой
- **В сигнатуру для перегрузки не входит возвращаемый тип!**
- ```
  int f(double a)
  double f(double a)
  // ошибка, т.к. уже есть метод с такой сигнатурой
  ```
  ```

Полиморфизм

- **Полиморфизм** – свойство, при котором сущности с одинаковым интерфейсом ведут себя по-разному
- Здесь под **интерфейсом** понимается то, что выставляется наружу, то есть то, с чем можно взаимодействовать
- Например, если у класса есть публичные методы и поля, то они и составляют интерфейс класса. Это то, к чему могут обратиться другие
- Перегрузка методов является одним из вариантов полиморфизма – методы с одинаковым именем, а, возможно, и даже одинаковым числом аргументов, ведут себя по-разному в зависимости от порядка и типов переданных аргументов

Методы с произвольным количеством аргументов

- Можно создавать методы, которые принимают любое количество аргументов одного типа
- ```
  public static double getAverage(double… numbers) {
      double sum = 0.0;
      for (double e : numbers) {
          sum += e;
      }
      return sum / numbers.length;
  }
  ```
- Как можно вызывать:

  ```
  getAverage(5);     // 5
  getAverage(2, 4); // 3
  ```

Переданные аргументы доступны как массив
  ```

Методы с произвольным количеством аргументов

- Использовать … в списке параметров метода можно только один раз
- При этом такой параметр обязательно должен быть последним в списке аргументов
- ```
  public static double f(int a, double… numbers) {
      // OK
  }
  ```
- ```
  public static double f(double… numbers, int a) {
      // Ошибка компиляции, double… numbers
      // должен быть последним параметром
  }
  ```

Методы с произвольным количеством аргументов

- На самом деле … это просто красивый синтаксис для методов, принимающих массив, который позволяет просто перечислить элементы массива при их передаче в метод
- Т.е. это аналогично:
- ```
  public static double getAverage(double[] numbers) {
      // код
  }
  ```
- Если объявить такой метод и с …, то компилятор выдаст ошибку, что метод с аргументом double[] уже объявлен
  ```

Методы с произвольным количеством аргументов

- ```
  public static double getAverage(double… numbers) {
      // код
  }
  ```

- Как и в метод, принимающий массив при помощи обычного синтаксиса, в методы с … можно передавать null и обычные массивы:
- getAverage(); // вызов от пустого массива
- getAverage(null); // вызов от null
- getAverage(new double[] { 1.0, 3.0 }); // 2
  ```

Enums (енумы, перечисления)

- **Enum** – это специальный вид класса, который позволяет задать набор допустимых значений-констант
- Enum’ы помогают обеспечить контроль типов по сравнению с обычными статическими константами, т.е. не позволяют передать в метод что-то ещё, кроме значений этого enum’а
- Enum’ы следует использовать практически всегда когда какой-то из параметров метода может принимать значения только из определенного набора
- Статья: http://www.quizful.net/post/java_enums

Enums

- Хотим хранить направления движения – Север, Юг, Запад, Восток
- Мы могли бы воспользоваться типом String или int чтобы объявить константы:
- ```
  public class Direction {
      public static final int SOUTH = 0;
      public static final int WEST = 1;
      public static final int EAST = 2;
      public static final int NORTH = 3;
  }
  ```
- Если какому-то методу потребуется направление в качестве параметра, то мы объявим функцию так:
  ```
  public void move(int direction, int offset) {
      // код
  }
  ```
  ```

Enums

- ```
  public class Direction {
      public static final int SOUTH = 0;
      public static final int WEST = 1;
      public static final int EAST = 2;
      public static final int NORTH = 3;
  }
  ```
- ```
  public void move(int direction, int offset) {
      // код
  }
  ```
- Проблема состоит в том, что те, кто использует наш код, могут передать в метод не только наши константы, но и просто число 10
- ```
  move(10, 1000); // число 10 не является направлением!
  ```
  ```

Enums

- public enum Direction {
  SOUTH, WEST, EAST, NORTH
  }
- Теперь метод можно объявить так:
- public void move(Direction direction, int offset) {
  // код
  }
- Туда не могут передать что-то недопустимое, можно передавать только значения enum’а Direction:
- move(Direction.SOUTH, 100);

Элементы enum’а доступны глобально

Полезные методы enum’ов

- ```
  enum Direction {
      SOUTH, WEST, EAST, NORTH
  }
  ```
- Все enum’ы неявно наследуются от класса java.lang.Enum и поэтому получают всего его методы
- Direction direction = Direction.SOUTH;

1. direction.toString() // SOUTH, но можно переопределить
2. direction.name() // SOUTH
3. direction.ordinal() // 0 - порядковый номер при
   // объявлении
  ```

Полезные методы enum’ов

- ```
  enum Direction {
      SOUTH, WEST, EAST, NORTH
  }
  ```

1. ```
   Direction.values();  // массив из всех значений enum’а
   ```
2. ```
   Direction d = Direction.valueOf(“SOUTH”);
   // получение экземпляра по имени
   // если нет значения с таким именем, то ошибка
   ```
   ```

Члены enum’ов

- Enum’ы в Java более сложны, чем во многих других языках – enum может иметь конструктор и методы и т.д.
- Пример – создадим enum для планет солнечной системы

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6);

    private final double mass;
    private final double radius;
    private static final double G = 6.673E-11;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double getGravity() {
        return G * mass / (radius * radius);
    }
}
```

Вызов метода:
Planet.VENUS.getGravity()

Статический инициализатор

- Кроме полей, методов и конструкторов, объявление класса может содержать **статический инициализатор**
- **Статический инициализатор** – это метод, который выполняется 1 раз перед первым использованием класса, и который может работать только со статическими полями класса
- Обычно используется, чтобы инициализировать статические поля класса
- Вообще, их можно делать сколько угодно в одном классе, они выполнятся в порядке объявления, но так обычно не делают

Статический инициализатор

- Обычно используется чтобы заполнить статические поля класса

- ```
  public class Currencies {
      private static final String[] currencies;

      static {
          currencies = new String[] { “RUB”, “USD” };
          // либо можем загрузить из файла
      }
  }
  ```
  ```

Статический инициализатор

- Когда мы присваиваем статическому полю значение сразу при объявлении поля, то на самом деле оно выполняется в статическом инициализаторе

- ```
  public class Currencies {
      private static final int CONSTANT = 1;
  }
  ```
  ```

Инициализатор экземпляра

- Это код, который выполняется при создании объекта
- Компилятор просто копирует этот код во все конструкторы, сразу после вызова конструктора-родителя, перед выполнением основного кода конструктора
- ```
  public class MyClass {
      private String field;

      {
          field = “Hello”;
      }
  }
  ```
- Используется, чтобы сократить дублирование кода между конструкторами
  ```

Инициализатор экземпляра

- Если мы сразу заполняем нестатические поля при объявлении, то этот код выполняется в инициализаторе экземпляра
- ```
  public class MyClass {
      private String field = “Hello”;
  }
  ```
- Обычно, так стараются не делать, а все писать в конструкторах, чтобы весь код инициализации экземпляра был в одном месте
  ```

Домашнее задание «Shapes»

- Реализовать задачу Shapes

Домашнее задание «Vector»

- Начать делать Vector