Лекция 5. Переопределение equals, hashCode, toString

Методы класса Object

- Как мы помним, все классы в Java в конечном итоге наследуются от Object, а значит, получают его методы

- Некоторые методы:

```
public boolean equals(Object obj)  // сравнение
public int hashCode()              // хэш-код
public String toString()           // преобразование в строку
```

Метод toString

Метод toString

- Метод **toString**() предназначен для преобразования объекта в строку
- По умолчанию для наших классов этот метод работает так:
- MyClass o1 = new MyClass();
  System.out.println(o1.toString()); // MyClass@3930015a
- Т.е. получается строка, которая начинается с имени класса
- Дальше идет @
- А дальше идет последовательность, которая зависит от **хэш-кода** объекта (см. далее в лекции)
- Более полное описание здесь: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString()

Метод toString

- Чтобы метод **toString** выводил что-то более осмысленное, его можно переопределить самим любым нужным нам образом
- Например, вывести поля объекта

- У многих стандартных классов **toString** уже переопределен хорошим образом
- Но у массивов он не переопределен:
- ```
  int[] a = new int[10];
  System.out.println(a.toString());  // [I@13790eb
  ```
  ```

Пример реализации toString

- Метод **toString**() можно определить как хочется
- ```
  public class Vector {
      private int[] vector;

      public Vector(int[] vector) { this.vector = vector;  }

      @Override
      public String toString() {
          return Arrays.toString(vector);
      }
  }
  ```

Arrays – это стандартный класс в Java

В нем есть методы для сортировки, бинарного поиска, копирования, заполнения массивов

- ```
  int[] a = { 1, 2, 3, 4 };
  Vector v = new Vector(a);
  System.out.println(v);   // [1, 2, 3, 4]
  ```

Метод toString

- Пусть у нас есть такой объект:
  - `Vector v = new Vector(new int[] { 1, 3, 5 });`
- Метод **toString** можно вызвать самим явно:
  - `System.out.println(v.toString()); // [1, 3, 5]`
- Также некоторые стандартные функции сами вызывают **toString**:
  - `System.out.println(v); // [1, 3, 5]`
- Метод **toString**() вызывается автоматически при конкатенации строки и объекта:
  - `System.out.println("vector = " + v);`
    `// vector = [1, 3, 5]`

Объект преобразовался в строку, строки конкатенируются

Сравнение ссылок. Метод equals

Операторы сравнения для объектов

- Как мы помним, в Java операторы == и != для объектов сравнивают ссылки, а не проверяют равенство объектов
- То есть в Java
  объект a == объект b
  тогда и только тогда, когда это один и тот же объект в памяти
- При этом не важен тип ссылки, важно что ссылки указывают на один и тот же объект:

```
A a = new A();
Object b = a;
System.out.println(a == b); // true, это тот же объект
```

Операторы сравнения для объектов

- ```
  Scanner s = new Scanner(System.in);
  String a = “123”;
  String b = s.nextLine();          // пусть вводят “123”
  System.out.println(a == b);   // false
  ```
- Но:
  ```
  String a = “123”;
  String b = “123”;
  System.out.println(a == b);   // будет true!
  ```
- Дело в том, что компилятор Java оптимизирует код при компиляции, и в исполняемом коде он будет использовать один и тот же объект в памяти для a и b
  ```

Операторы сравнения для объектов

- ```
  String a = “123”;
  String b = “123”;
  System.out.println(a == b);   // будет true, как так?
  ```
- Дело в том, что компилятор Java оптимизирует код при компиляции, и в исполняемом коде он будет использовать один и тот же объект в памяти для a и b
- Результирующий код будет работать так:
- ```
  String a = “123”;
  String b = a;
  System.out.println(a == b);   // будет true
  ```

Метод equals

- А вот метод **equals**(Object o) предназначен для проверки на равенство содержимого объектов
- ```
  Scanner s = new Scanner(System.in);
  String a = “123”;
  String b = s.nextLine();          // пусть вводят “123”
  System.out.println(a == b);   // false
  ```
- ```
  Scanner s = new Scanner(System.in);
  String a = “123”;
  String b = s.nextLine();                  // пусть вводят “123”
  System.out.println(a.equals(b));    // true, как надо
  ```

Метод equals в классе Object

- По умолчанию, в классе Object, метод **equals**(Object o) просто проверяет равенство ссылок при помощи ==
- Многие стандартные классы, такие как String, переопределяют метод **equals**, чтобы он сравнивал содержимое объектов
- Поэтому в своих классах, если мы хотим сравнивать их объекты, нужно переопределить метод **equals**

Как переопределять метод equals?

- Есть 2 варианта:
  - Через проверку на равенство классов
  - Через instanceof

- Рекомендуется вариант 1, позже рассмотрим почему

Как переопределять метод equals?

```
public class Pair {
    private int first;
    private int second;

    public Pair(int first, int second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public boolean equals(Object o) {
        // реализация
    }
}
```

Через равенство классов

```
public class Pair {
    private int first;
    private int second;

    public boolean equals(Object o) {
        // проверили что передали сам объект
        if (o == this) return true;
        // отсеяли null и объекты других классов
        if (o == null || o.getClass() != getClass()) return false;
        // привели объект к Pair
        Pair p = (Pair) o;
        // проверили равенство ссылок и полей
        return first == p.first && second == p.second;
    }
}
```

Возможно, не все поля должны участвовать в equals, например, если некоторые поля являются вспомогательными

Фигурные скобки опущены только потому что не влезли

Через instanceof

```
public class Pair {
    private int first;
    private int second;

    @Override
    public boolean equals(Object o) {
        // отсеяли null и объекты других типов
        if (!(o instanceof Pair)) return false;
        // привели объект к Pair
        Pair p = (Pair) o;
        // проверили равенство ссылок и полей
        return (o == this) || (first == p.first && second == p.second);
    }
}
```

Фигурные скобки опущены только потому что не влезли

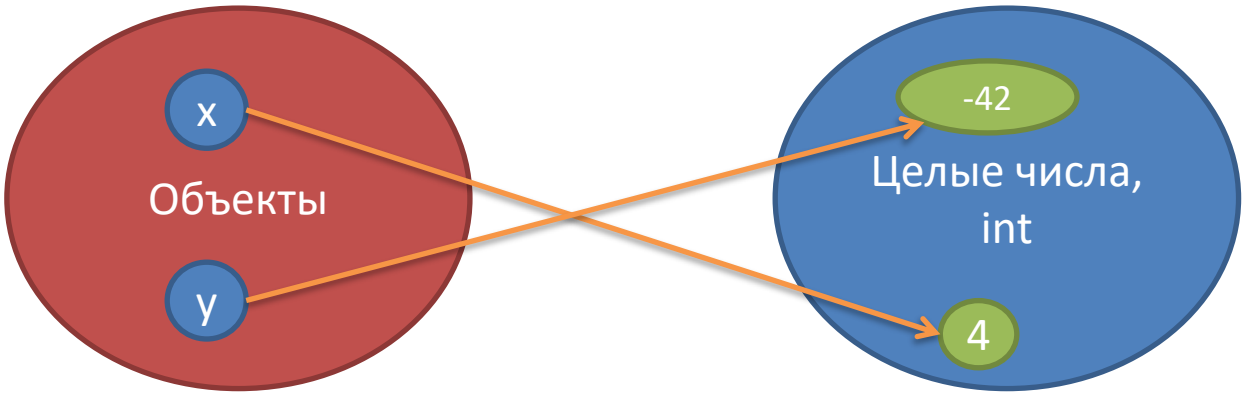
Как переопределять метод equals?

- Если проверять через instanceof , то смогут сравниваться объекты базового класса и объекта класса-наследника
- Пусть класс B наследуется от A, оба класса реализуют **equals** через instanceof
- ```
  A a = new A();
  B b = new B();
  a.equals(b)   // true
  b.equals(a)   // false
  ```
- Т.е. тут нарушена симметричность, что в некоторых случаях плохо. Надежнее реализовать через проверку, что класс в точности совпадает
  ```

Хэш-функция. hashCode

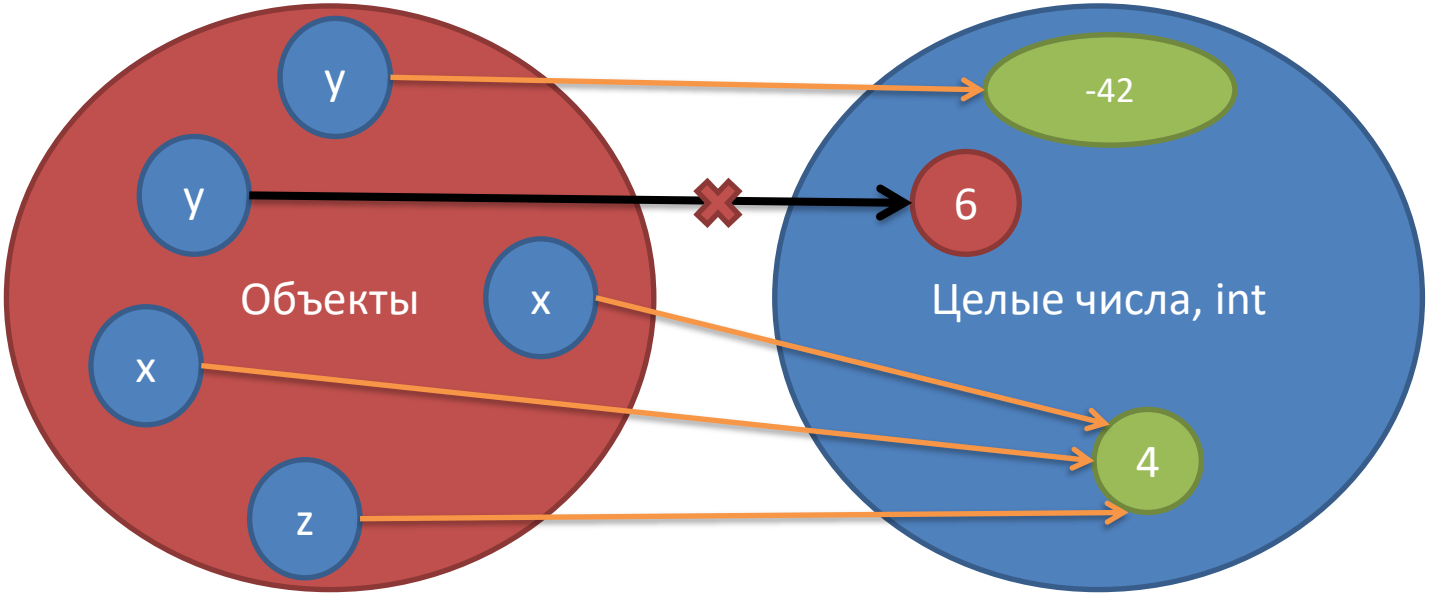
Хэш-функция

- **Хэш-функция** – это функция, которая принимает объект и, используя данные объекта, вычисляет целое число, и обладает некоторыми свойствами
(см. через несколько слайдов)
- Результат применения хэш-функции к объекту называется **хэш-код** или просто **хэш**
- В Java за вычисление хэш-функции отвечает метод **hashCode**()



Свойства хэш-функции

- Хэш функция должна быть связана со сравнением объектов следующим образом:
  - если объекты равны, то их хэши должны быть равны
  - если объекты не равны, то их хэши желательно должны быть разными, но это не обязательно



Хэш-функция

- Пример хэш-функции
-
```
public class A {
    private int a;

    @Override
    public int hashCode() {
        return a;    // все формальности соблюдены
    }                // это int, для равных объектов он совпадает

    public boolean equals(Object obj) {
        if (!(obj instanceof A)) return false;
        return (obj == this) || (a == ((A)obj).a);
    }
}
```

Свойства хэш-функции

- Хэш-функция должна быть связана со сравнением объектов следующим образом:
  - если объекты равны, то их хэши должны быть равны
  - если объекты не равны, то их хэши желательно должны быть разными, но это не обязательно
  - если объект не менялся, то хэш не должен меняться
- Получается следующее:
  - если хэши разные, то объекты точно разные
  - если хэши равны, то не факт, что равны объекты, надо проверять дальше при помощи сравнения
- Так как хэш-функция связана со сравнением, то методы **hashCode** и **equals** всегда надо переопределять вместе

Хэш-функция

```
public class A {
    private int a;

    @Override
    public int hashCode() {
        return a;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof A)) return false;
        return (obj == this) || (a == ((A)obj).a);
    }
}
```

Методы hashCode и equals оба переопределены. Свойства выполняются

Как написать hashCode?

```
public class A {
    private int a;
    private double b;
    private B c;
    private double[] d;

    public int hashCode() {
        final int prime = 37;
        int hash = 1;
        hash = prime * hash + a;
        hash = prime * hash + Double.hashCode(b);
        hash = prime * hash + (c != null ? c.hashCode() : 0);
        hash = prime * hash + Arrays.hashCode(d);
        return hash;
    }
}
```

В качестве начального значения берут число != 0

Потом поле за полем, делают
hash = prime * hash + (хэш от этого поля)

prime – некоторое нечетное простое число, например, 37

Arrays.hashCode – правильно вычисляет хэш-код от массива

Зачем нужна хэш-функция?

- Позволяет быстро определить, что объекты не равны.
  Если хэши разные, то объекты – разные
  - Но если хэши совпали, то надо проверять, что объекты равны
  - Хэш-функция обычно вычисляется быстро, и это дешевле, чем выполнить полное сравнение
- На хэш-функции основана структура данных **хэш-таблица**, которая позволяет осуществлять быстрый поиск
- Хэш-таблица будет рассмотрена в следующих лекциях

Records

Records

- Начиная с Java 16 (начиная с 14 была preview версия), появился новый вид классов – **records**
- https://docs.oracle.com/en/java/javase/16/language/records.html
- **Record** позволяет кратко создать класс с неизменяемыми полями, в котором сразу будут реализованы конструктор, геттеры, а также **equals**, **hashCode**, **toString**
- В основном **records** используются для передачи данных (т.е. чтобы сделать класс с полями без дополнительной логики)

Records

- public record Rectangle(double width, double height) {
  }
- Т.е. вместо слова class указываем record
- И в скобках указываем аргументы для конструктора
  - Сами скобки обязательные даже если нет аргументов
  - Этот конструктор называется **каноническим**
- В самом простом варианте это весь наш код – сам код конструктора, поля, геттеры, **equals**, **hashCode**, **toString** генерируются сами

Records

- По смыслу это эквивалентно такому коду:
- public final class Rectangle extends Record {
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double width() { return width; }
    public double height() { return height; }

    /* также реализованы **equals**, **hashCode**, **toString** */
}

Наследуется от Record

Заметим, что поля final

Имена геттеров начинаются не с get

Изменение реализации конструктора

- При желании в **record** можно явно написать канонический конструктор
- Тогда будет использоваться своя реализация вместо стандартной
- Один из вариантов - использовать полный синтаксис, как для обычных классов

Изменение реализации конструктора

- Либо можно использовать особый краткий синтаксис:
-
```
public record Rectangle(double width, double height) {
    public Rectangle {
        if (width <= 0) {
            throw new IllegalArgumentException("Width must be > 0."
                + Current value: " + width);
        }

        if (height <= 0) {
            throw new IllegalArgumentException("Height must be > 0."
                + Current value: " + height);
        }
    }
}
```

У конструктора не пишутся скобки и аргументы

Доступ к аргументам идет по их именам

Не нужно писать сам код присваивания полей, компилятор подставляет его сам

Records

- Также можно самим явно реализовать геттеры, **equals**, **toString**, **hashCode**
- В records нельзя объявлять нестатические поля.
  Все поля берутся только из объявления в заголовке record'а
- Records не могут наследоваться (потому что уже наследуются от Record) и нельзя наследоваться от них (потому что по факту они final)

Документация про новые фичи языка

- Рекомендую, в целом, прочитать документацию по изменениям языка Java:
- https://docs.oracle.com/en/java/javase/18/language/java-language-changes.html
- В курсе рассматриваются не все эти фичи, но многие из них небольшие и простые, поэтому изучите их самостоятельно
- Посмотрите следующие темы:
  - https://docs.oracle.com/en/java/javase/17/language/sealed-classes-and-interfaces.html
  - https://docs.oracle.com/en/java/javase/16/language/pattern-matching-instanceof-operator.html
  - https://docs.oracle.com/en/java/javase/15/language/text-blocks.html