Программирование
1 семестр

ИТМО

Наследование

Базовые принципы ООП

ИТМО

- Абстракция
- Инкапсуляция
- **Наследование**
- Полиморфизм

- Inheritance
- Создание новых классов на основе существующих с сохранением или расширением функциональности. При этом реализуется иерархия классов, связанных отношением "is-a".

Rectangle

```
class Rectangle {
  private double width, height;
  public Rectangle(double w, double h) {
    this.width = w;
    this.height = h;
  }
  public double area() {
    return width * height;
  }
  public String descr() {
    return width + "*" + height;
  }
}
```

Square

```
class Square {
  private double side;
  public Square(double side) {
    this.side = side;
  }
  public double area() {
    return side * side;
  }
  public String descr() {
    return side + "*" + side;
  }
}
```

Square

ИТМО

- Квадрат - это прямоугольник, у которого стороны равны

```
class Rectangle {
  private double width, height;
  public Rectangle(double w, double h) {
    this.width = w;
    this.height = h;
  }
  public double area() {
    return width * height;
  }
  public String descr() {
    return width + "*" + height;
  }
}
```

Square

ИТМО

- Квадрат - это прямоугольник, у которого стороны равны

```
class Rectangle {
  private double width, height;
  public Rectangle(double w, double h) {
    this.width = w;
    this.height = h;
  }
  public double area() {
    return width * height;
  }
  public String descr() {
    return width + "*" + height;
  }
}
```

```
class Square extends Rectangle {
  public Square(double side) {
  // вызов конструктора Rectangle
    super(side, side);
  }
}
```

Square

ИТМО

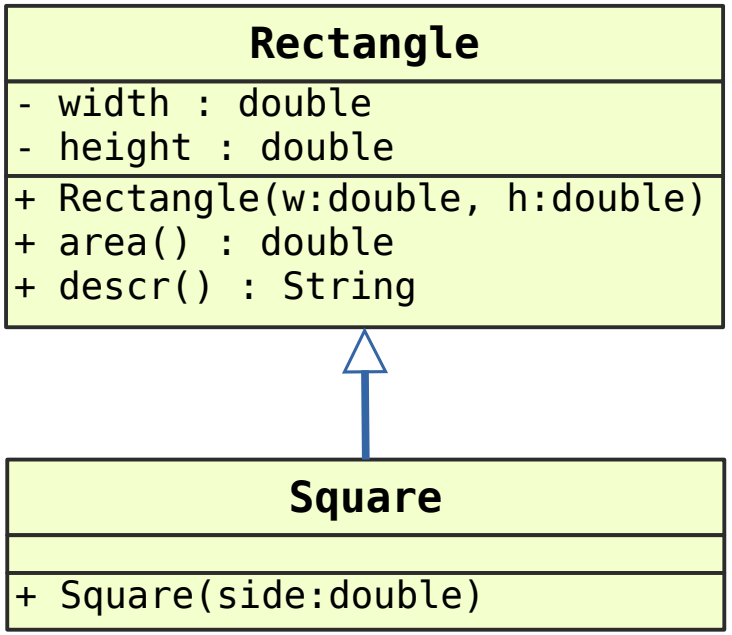
- Квадрат - это прямоугольник, у которого стороны равны

```
class Rectangle {
  private double width, height;
  public Rectangle(double w, double h) {
    this.width = w;
    t
  }
  pub
    r
  }
  pub
    r
  }
}
```

```
class Square extends Rectangle {
  public Square(double side) {
  // вызов конструктора Rectangle
    super(side, side);
  }
}
```

```
class Main {
  Square[] squares = {
    new Square(3.5),
    new Square(5.0)
  };
  for (Square sq : squares) {
    System.out.println("Квадрат: " + sq.descr());
    System.out.println("Площадь: " + sq.area());
}
```

UML - наследование

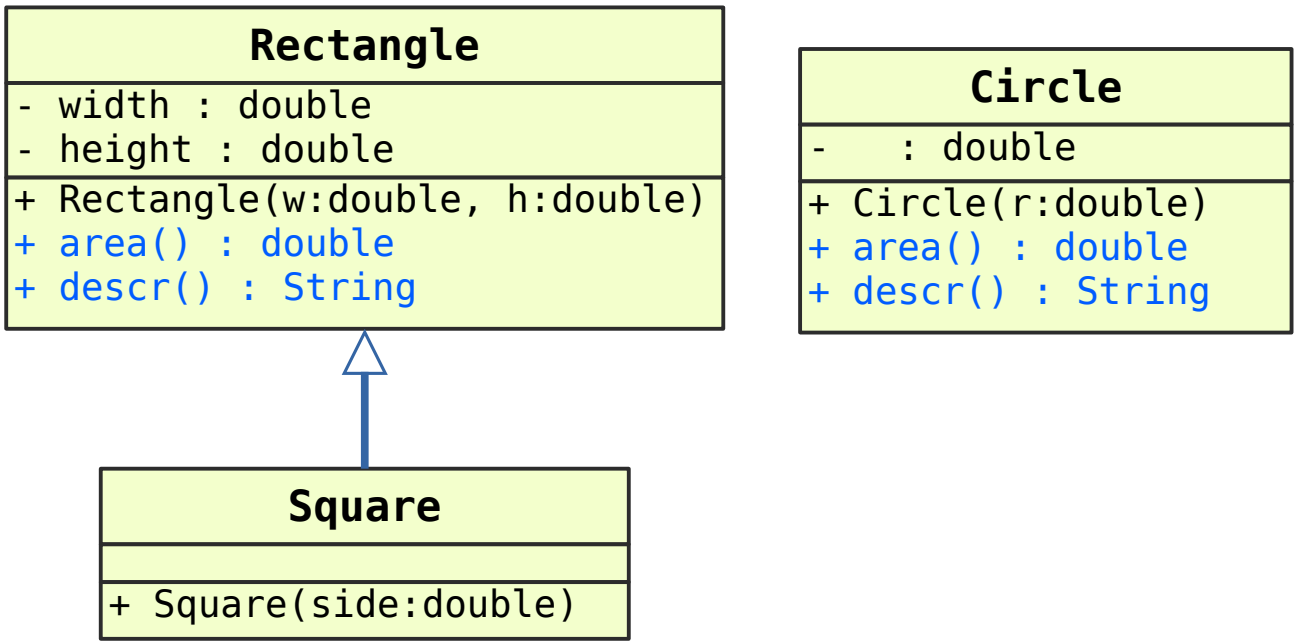


Наследование

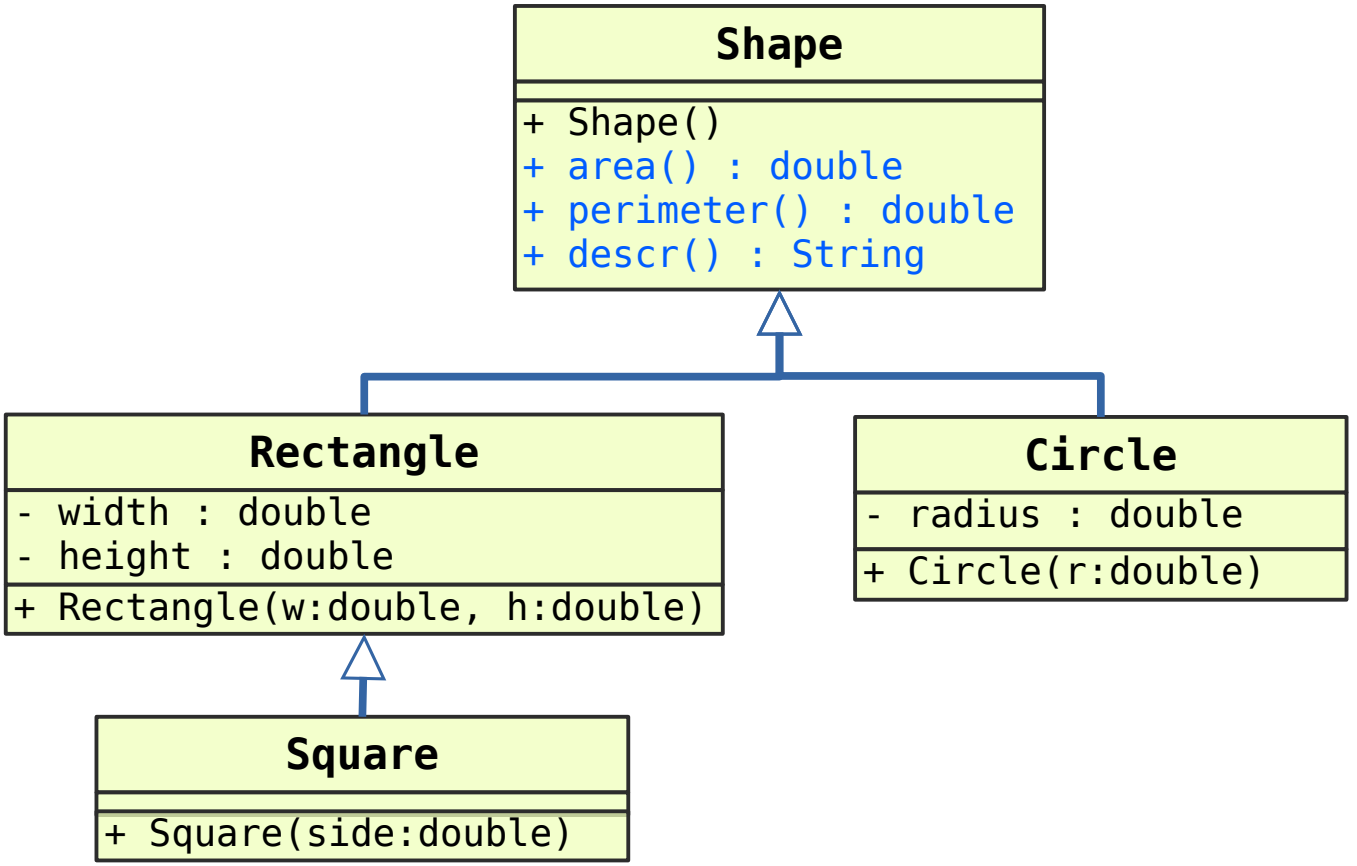
- Parent class (предок)
- Superclass (суперкласс)
- Base class (базовый)

- Child class (потомок)
- Subclass (подкласс)
- Extended class (расширенный)

Дублирование кода



UML - наследование

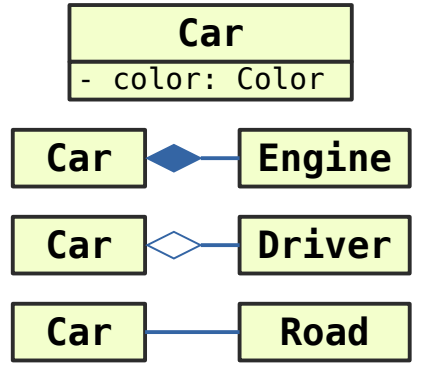


Отношения is-a и has-a

ИТМО

- is-a (наследование)
  - Dog IS-A Mammal
  - любая собака - млекопитающее
- направленное отношение
  - не любое млекопитающее - собака
- специализация
  - собака может лаять
  - собака издает звук "Гав!"
- иерархия

- has-a (принадлежность)
  - Dog HAS-A Head
  - у любой собаки есть голова
- виды
  - свойство
  - композиция
  - агрегация
  - ассоциация



Модификаторы для полей, методов, классов

• private	• Доступ только своим (внутри класса)
•	• Доступ классам того же пакета
• protected	• Доступ классам пакета и наследникам
• public	• Доступ всем (почти)

Наследование: доступ к полям и методам

ИТМО

```
package my;
public class Base {
  private int privField = 1;
  int packageField = 2;
  protected int protField = 3;
  public int publField = 4;
}
```

```
package other;
class Extended extends Base {
  int priv = super.privField;
  int pack = super.packageField;
  int prot = super.protField;
  int publ = super.publField;
}
```

Наследование: сокрытие поля (hiding)

ИТМО

```
package my;
public class Base {
  protected int protField = 3;
  public int publField = 4;
  public static int statField = 5;
}
```

```
package other;
class Extended extends Base {
  protected float protField = 7.0;
  public float publField = 8.0;
  public static int statField = 9;
  ...
  print(this.protField); // 7.0
  print(super.protField) // 3
  print(this.publField); // 8.0
  print(super.publField) // 4
  print(statField);      // 9
  print(Base.statField)  // 5
}
```

Наследование: доступ к полям через методы

ИТМО

```
package my;
public class Base {
  private int privField = 1;
  protected int getPriv() {
    return privField;
  }
  private static int statField = 2;
  protected static int getStat() {
    return statField;
  }
}
```

```
package other;
class Extended extends Base {
  int priv = super.privField;
  int privField = getPriv(); // 1


  private static int statField = 3;
  public static int getStat() {
    return statField;
  }
  ...
  print(getStat());       // 3
  print(Base.getStat()); // 2
}
```

Переопределение методов (overriding)

ИТМО

- Переопределение методов
  - Создание в подклассе методов с тем же именем и набором параметров, как в суперклассе
  - Меняется поведение наследника - не должен нарушаться контракт базового класса

Переопределение и перегрузка методов

ИТМО

- Переопределение методов (overriding)
  - Метод наследника имеет то же имя и **тот же набор параметров**
  - Меняется поведение наследника
- Перегрузка методов и конструкторов (overloading)
  - Метод или конструктор имеет то же имя и **другие параметры**
  - Позволяет одинаково обрабатывать разные типы данных

Статический метод с тем же именем

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(Base.statVal()); // 3
System.out.println(Ext.statVal());  // 4
```

Сокрытие статического метода (hiding)

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(base.statVal()); // 3
System.out.println(ext.statVal());  // 4
System.out.println(baseExt.statVal());
```

Вызов статического метода через объект

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(base.statVal()); // 3
System.out.println(ext.statVal());  // 4
System.out.println(baseExt.statVal());  // 3
```

Переопределение методов экземпляра

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(base.getVal()); // 1
System.out.println(ext.getVal());  // 2
System.out.println(baseExt.getVal());
```

Переопределение методов экземпляра

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(base.getVal()); // 1
System.out.println(ext.getVal());  // 2
System.out.println(baseExt.getVal()); // 2
```

Переопределение - расширение доступа

ИТМО

```
public class Base {
  protected int getVal() {
    return 1;
  }
  protected static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

Переопределение - сужение доступа

ИТМО

```
public class Base {
  protected int getVal() {
    return 1;
  }
  protected static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  private int getVal() {
    return 2;
  }
  private static int statVal() {
    return 4;
  }
}
```

```
Error:
getVal() in Ext cannot override getVal() in Base
attempting to assign weaker access privileges; was protected
```

Аннотация @Override

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  @Override public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(baseExt.getVal()); // 2
```

Аннотация @Override - бесполезна?

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getVal() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(baseExt.getVal()); // 2
```

Аннотация @Override - бесполезна?

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getval() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(baseExt.getVal()); //
```

Аннотация @Override - если ошибка?

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getval() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(baseExt.getVal()); //
```

Аннотация @Override - если ошибка?

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  public int getval() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Base base = new Base();  Ext ext = new Ext(); Base baseExt = new Ext();

System.out.println(baseExt.getVal()); // 1
```

Аннотация @Override - полезна!

ИТМО

```
public class Base {
  public int getVal() {
    return 1;
  }
  public static int statVal() {
    return 3;
  }
}
```

```
public class Ext extends Base {
  @Override public int getval() {
    return 2;
  }
  public static int statVal() {
    return 4;
  }
}
```

```
Error:
method does not override or implement a method from a supertype
@Override public int getval() {
```

Поля и методы при наследовании

ИТМО

- Наследуются:
  - поля и методы с модификаторами public и protected
  - поля и методы без модификатора доступа, если подкласс и суперкласс в одном пакете
- Не наследуются:
  - поля и методы с модификатором private (не доступны)
  - конструкторы
  - блоки инициализации

Поля и методы при наследовании

ИТМО

- Если имя поля в подклассе и в суперклассе одинаковое
  - Поле подкласса скрывает поле суперкласса
  - Статическое поле суперкласса доступно по имени класса

    ```
    BaseClass.staticField
    ```

  - Нестатическое поле суперкласса доступно с помощью super

    ```
    super.instanceField
    ```

  - Скрытое поле может иметь другой тип

Поля и методы при наследовании

ИТМО

- Если сигнатура (имя и параметры) статического метода в подклассе и в суперклассе одинаковая
  - Статический метод подкласса скрывает метод суперкласса
  - Статический метод суперкласса доступен по имени класса

```
BaseClass.staticMethod()
```

Поля и методы при наследовании

ИТМО

- Если сигнатура (имя и параметры) нестатического метода в подклассе и в суперклассе одинаковая
  - Метод подкласса переопределяет метод суперкласса
  - Метод суперкласса доступен с помощью super

    ```
    super.instanceMethod()
    ```
  - Доступ к переопределенному методу нельзя уменьшать
    - Если в суперклассе public - в подклассе тоже public
    - Если в суперклассе protected - в подклассе protected или public

Приведение типов

ИТМО

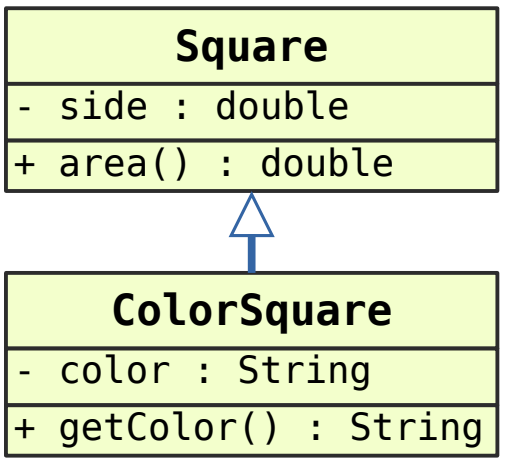
- Возможность вызвать метод определяется по типу ссылки
- Ссылку нельзя привести к произвольному типу
- Ссылку можно привести к более общему типу
- Ссылку можно привести к реальному типу объекта
- Оператор instanceof проверяет тип объекта

Ссылки и объекты

ИТМО

```
class Square {
  private double side;
  public Circle(double s) {
    this.side = s;
  }
  public double area() {
    return side * side;
  }
}
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```



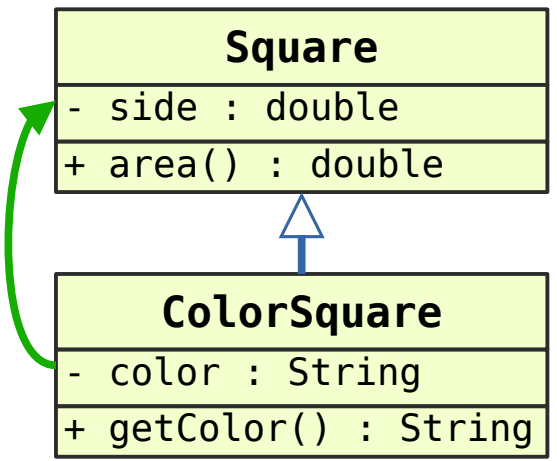
Оператор instanceof

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```

```
ColorSquare s = new ColorSquare(1.5, "red");

s instanceof ColorSquare // true
s instanceof Square // true
s instanceof String // false
```



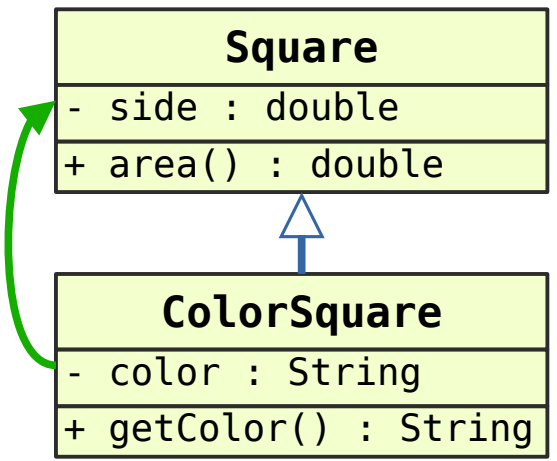
Приведение ссылочных типов (upcasting)

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```

```
ColorSquare s = new ColorSquare(1.5, "red");
Square x = s; // приведение к суперклассу
```



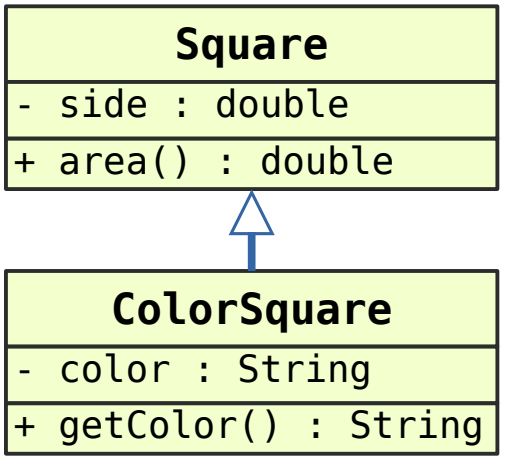
Приведение ссылочных типов (upcasting)

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```

```
ColorSquare s = new ColorSquare(1.5, "red");
Square x = s; // приведение к суперклассу
x.area(); // 2.25
x.getColor(); // Ошибка - нет такого метода
```



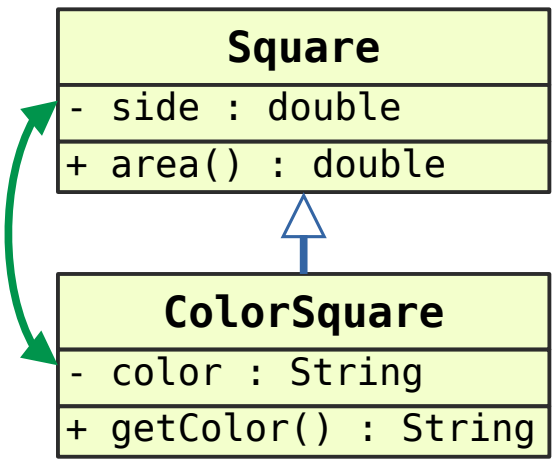
Приведение ссылочных типов (downcasting)

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```

```
ColorSquare s = new ColorSquare(1.5, "red");
Square x = s; // приведение к суперклассу
x.area(); // 2.25

ColorSquare y = (ColorSquare) x;
y.getColor(); // red
```



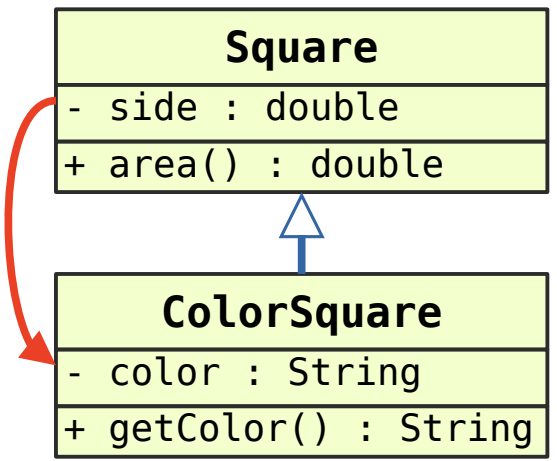
Приведение ссылочных типов (downcasting)

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
Square s = new Square(1.5);
Square x = s;
x.area(); // 2.25

ColorSquare y = (ColorSquare) x;
// ClassCastException
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```



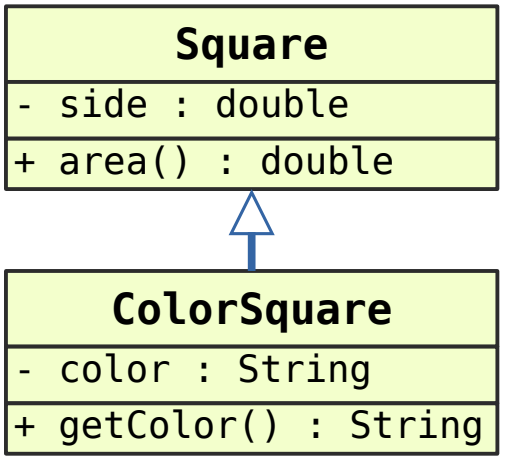
Приведение ссылочных типов

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double s, String color) {
    super(s); this.color = color;
  }
  public String getColor() { return color; }
}
```

```
ColorSquare s = new ColorSquare(1.5, "red");
Square x = s;
x.area(); // 2.25
if (x instanceof ColorSquare) {
  ColorSquare y = (ColorSquare) x;
  y.getColor(); } // OK!
```



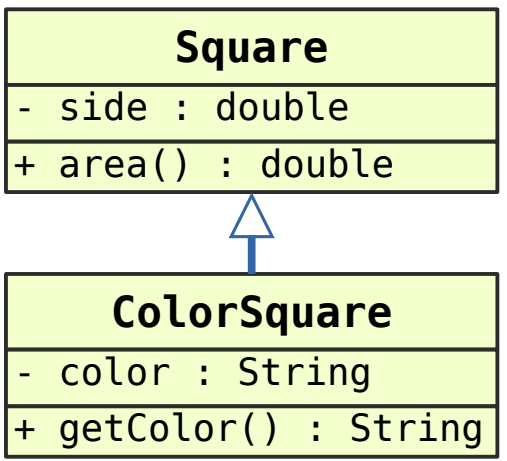
Pattern Matching - сопоставление шаблона

ИТМО

```
class Square {
  private double side;
  public Square(double s) {
    this.side = s;
  }
  public double area() {
    return Math.PI * side * side;
  }
}
```

```
ColorSquare s = new ColorSquare(1.5, "red");
Square x = s;
x.area(); // 2.25
if (x instanceof ColorSquare y) {

  y.getColor(); } // OK!
```

```
class ColorSquare extends Square {
  private String color = "Unknown";
  public ColorSquare(double r, String color) {
    super(r); this.color = color;
  }
  public String getColor() { return color; }
}
```



Что умеет пустой класс?

```
public class Empty { }
```

Что умеет пустой класс?

ИТМО

```
public class Empty { }
```

```
Empty obj = new Empty();

System.out.println(obj.equals(null));
// false

System.out.println(obj.hashCode());
// 1338668845

System.out.println(obj.toString());
// Empty@4fca772d

System.out.println(obj.getClass());
// class Empty
```

Что умеет пустой класс?

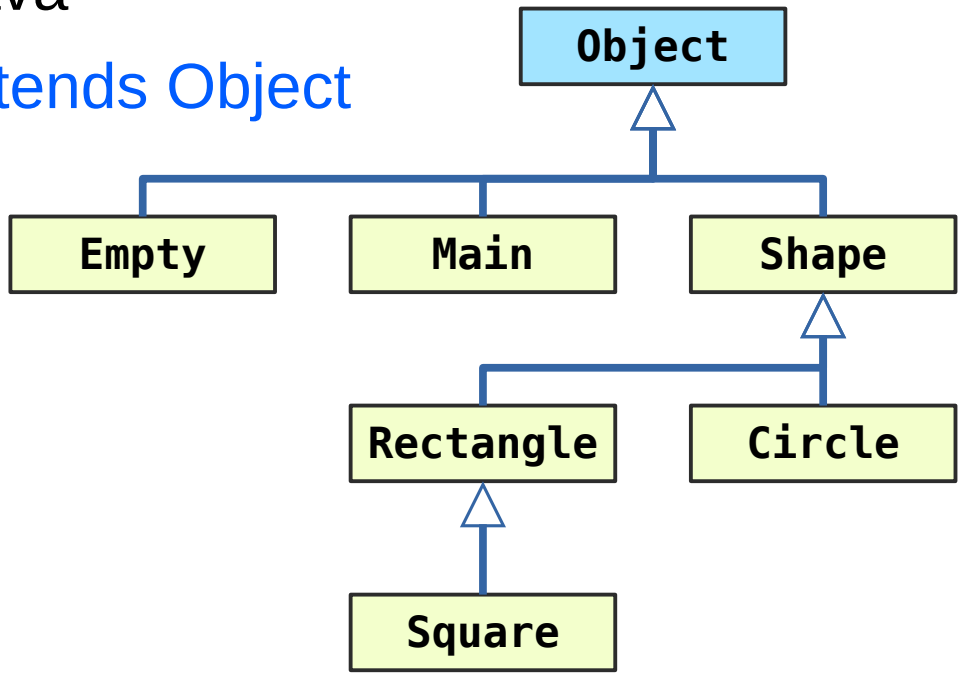
ИТМО

```
import java.lang.*;

public class Empty extends Object {
  public Empty() {
    super();
  }
}
```

```
Empty obj = new Empty();

System.out.println(obj.equals(null));
// false

System.out.println(obj.hashCode());
// 1338668845

System.out.println(obj.toString());
// Empty@4fca772d

System.out.println(obj.getClass());
// class Empty
```

Класс java.lang.Object

- Object - предок всех классов Java
- Если отсутствует extends = extends Object
- Методы класса Object:
  - public Class getClass()
  - public boolean equals()
  - public int hashCode()
  - public String toString()
  - prootected Object clone()
  - void wait(), void notify(), void notifyAll()



Метод Object.toString()

ИТМО

- Представление объекта в виде текстовой строки
  - Для Object - имя класса + @ + хэшкод

```
class Rectangle {
  public String toString() {
    return "Rectangle (" + width + " * " + height + ")";
  }
}
```

Метод Object.hashCode()

ИТМО

- Целое число, определяющее состояние объекта
  - Хэшкоды эквивалентных объектов **должны быть равными**
  - Хэшкоды различных объектов **не обязаны быть различными**
  - Для Object - обычно вычисляется из адреса и кэшируется

```
class Rectangle {
  public int hashCode() {
    return 65535 * width + height;
  }
}
```

Метод Object.equals()

ИТМО

- Сравнение объектов на эквивалентность
  - для Object возвращает obj1 == obj2 (сравнение ссылок)
  - true, если ссылка на один и тот же объект
- Рекомендуется переопределить в своем классе

```
Object obj = new Object();
Object other = new Object();
Object same = o1;
```

```
obj == other // false
obj == same  // true
```

```
obj.equals(other) // false
obj.equals(same)  // true
```

Метод Object.equals()

ИТМО

- Свойства отношения эквивалентности
  - рефлексивность: x.equals(x) == true
  - симметричность: x.equals(y) == y.equals(x)
  - транзитивность: x.equals(y) == true, y.equals(z) == true ⇒ x.equals(z) == true
  - консистентность: результат не должен меняться при последовательных вызовах с теми же значениями
  - если x != null, то x.equals(null) == false

Метод Object.equals()

ИТМО

- Можно сравнить значения значимых полей
- Остальные поля можно игнорировать

```
class Rectangle {
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Rectangle)) { return false; }
    Rectangle other = (Rectangle) o;
    return this.width == other.width && this.height == other.height;
  }
}
```

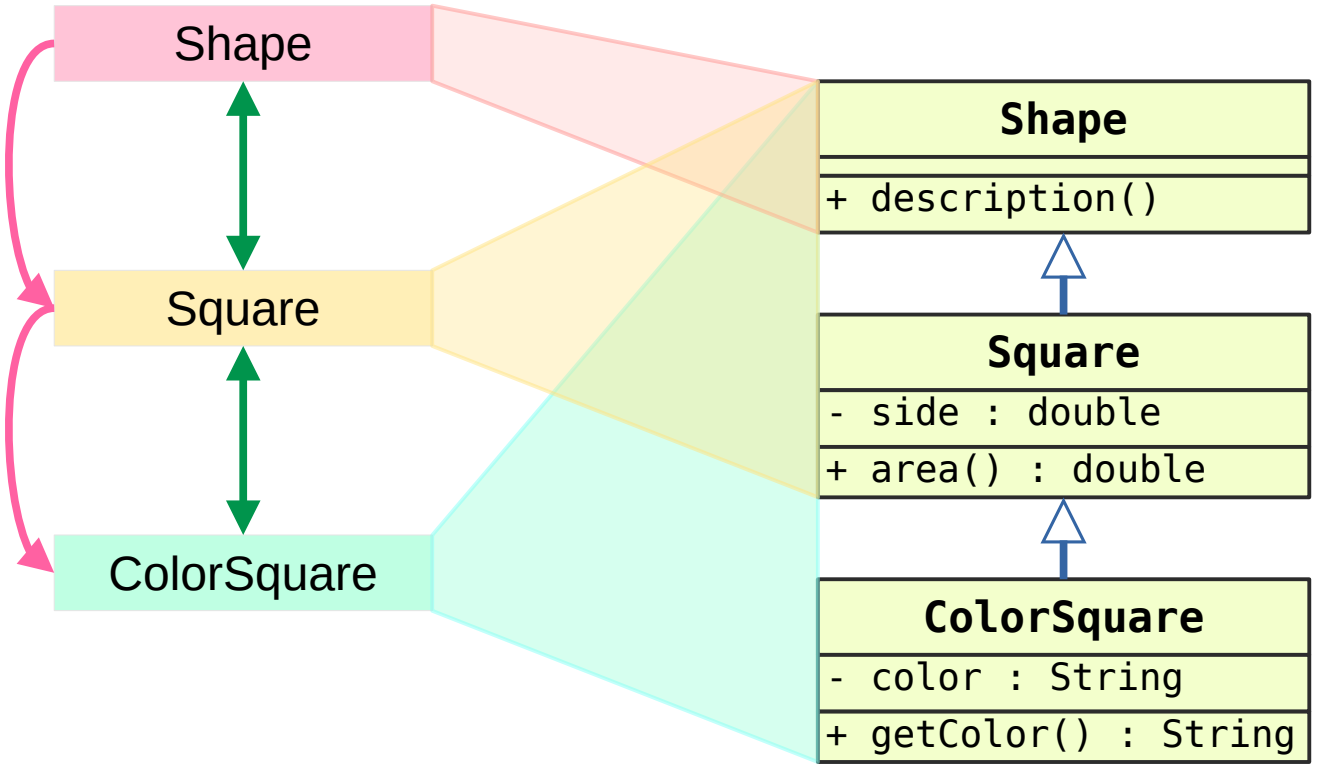
equals() и hashCode()

ИТМО

- equals() и hashCode() должны быть согласованными

```
class Rectangle {
  public int hashCode() {
    return Objects.hash(this.area(), this.perimeter());
  }
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Rectangle)) { return false; }
    if (o.hashCode() != hashCode()) { return false; }
    Rectangle other = (Rectangle) o;
    return this.area() == other.area() && this.perimeter() == other.perimeter();
  }
}
```

Иерархия и преобразование типов

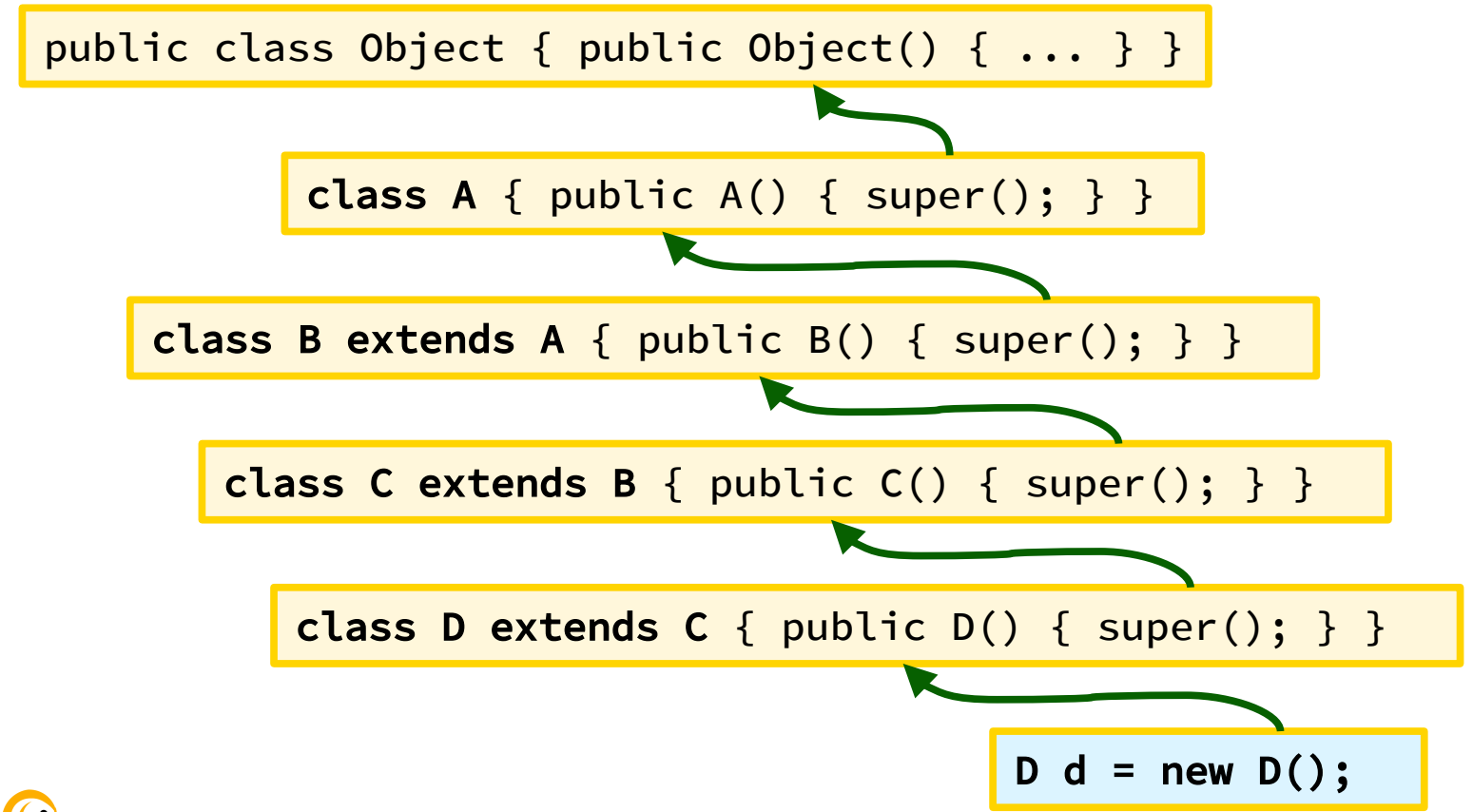


Конструктор по умолчанию

ИТМО

- Если в классе не задан конструктор
  - Компилятор добавит пустой конструктор без параметров
- Если в конструкторе нет явного вызова super() или this()
  - Компилятор добавит вызов super() без аргументов
- Если в суперклассе такого конструктора нет
  - Явно добавить его
  - Добавить вызов super() или this() с корректными аргументами

Цепочки конструкторов



Ключевое слово final

- Финальные классы
  - final для переменной - нельзя поменять значение
  - final для метода - нельзя переопределять
  - final для класса - нельзя наследовать

Ключевое слово sealed

ИТМО

- Запечатанные классы
  - final class - наследование полностью запрещено
  - не final class - наследование разрешено всем
  - sealed class - контроль наследования

```
public sealed class Shape permits Circle, Rectangle, Triangle { }

public final class Circle extends Shape { }
public sealed class Rectangle extends Shape permits Square { }
public non-sealed class Triangle extends Shape { }
```

Enum

ИТМО

- Тип - перечисление (неявно extends Enum)
- Список именованных констант
- Набор значений небольшой и не меняется

```
enum Month {
   JANUARY, FEBRUARY,
   MARCH, APRIL, MAY,
   JUNE, JULY, AUGUST,
   SEPTEMBER, OCTOBER,
   NOVEMBER, DECEMBER
}
```

```
Month current = Month.OCTOBER;

System.out.println(current)
// OCTOBER - не String
```

Enum - продвинутое перечисление

ИТМО

```
enum Month {
   JANUARY, FEBRUARY,
   MARCH, APRIL, MAY,
   JUNE, JULY, AUGUST,
   SEPTEMBER, OCTOBER,
   NOVEMBER, DECEMBER
}
```

```
Month current = Month.OCTOBER;

current.ordinal() // 9

current.name()    // "OCTOBER" (String)

Month.values()
/* { JANUARY, FEBRUARY, MARCH, APRIL,
MAY, JUNE, JULY, AUGUST, SEPTEMBER,
OCTOBER, NOVEMBER, DECEMBER } */

Month.valueOf("MAY") // Month.MAY
```

Enum - еще более продвинутое перечисление

ИТМО

```
enum Month {
  JANUARY(31),FEBRUARY(28),MARCH(31),APRIL(30),MAY(31),JUNE(30),
  JULY(31),AUGUST(31),SEPTEMBER(30),OCTOBER(31),NOVEMBER(30),DECEMBER(31);

  private int days;

  public Month(int days) {
    this.days = days;
  }

  public numberOfDays() {
    return days;
  {
}
```

Enum - еще более продвинутое перечисление

ИТМО

```
enum Month {
  JANUARY(31),FEBRUARY(28),MARCH(31),APRIL(30),MAY(31),JUNE(30),
  JULY(31),AUGUST(31),SEPTEMBER(30),OCTOBER(31),NOVEMBER(30),DECEMBER(31);

  private int days;

  public Month(int days) {
    this.days = days;
  }

  public numberOfDays() {
    return days;
  {
}
```

```
Month current = Month.OCTOBER;

current.numberOfDays() // 31
```

Record - запись

ИТМО

```
public record Point(int x, int y) {}
```

- Неизменяемый класс для данных
- Автоматически создаются:
  - приватные финальные поля
  - конструктор
  - геттеры
- Переопределяются методы класса Object

- Можно добавить
  - компактный конструктор
  - методы экземпляра
  - статические поля и методы
  - добавлять интерфейсы
- Нельзя:
  - наследовать
  - добавлять обычные поля

Record - что там есть?

ИТМО

```
public record Point(int x, int y) {}
```

```
Point p1 = new Point(2, 3);
Point p2 = new Point(2, 3);

System.out.println(p1.equals(p2));
// true
System.out.println(p1.hashCode());
// 65
System.out.println(p2.toString());
// Point[x=2, y=3]

int a = p1.x()-p2.x()+p1.y()-p2.y();
```

```
public final class Point extends Record{
  private final int x, y;
  public Point (int x, int y) {
    this.x = x; this.y = y;
  }
  public String toString() {
    return "Point[x="+ x +", y="+ y;
  }
  public boolean equals(Point p) {
    return p.x == x && p.y == y;
  }
  public int hashCode() {
    return x * 31 + y;
  }
  public final int x() { return x; }
  public final int y() { return y; }
}
```

Record - что можно добавить?

ИТМО

```
public record Rectangle(double width, double height) {
  public Rectangle {
    System.out.println("Constructor is executed");
  }
  public double area() {
    return width * height;
  }
  public double perimeter() {
    return 2 * (width + height);
  }
}
```

```
Rectangle r = new rectangle(2.5, 4.0);

System.out.println(r);
System.out.println("Area: "+r.area());
System.out.println("Perimeter: " +
                   r.perimeter());
```

Библиотека классов Java - immutable classes

ИТМО

- java.lang.String - immutable
- java.lang.StringBuilder - mutable

```
String x = "";
for (int i=0; i<10000000; i++) {
  x += " " + i;
}
// в каждой итерации – новый объект
```

```
StringBuilder sb = new StringBuilder();
for (int i=0; i<10000000; i++) {
  sb.append(" " + i);
}
String x = sb.toString();

// sb.insert()
// sb.delete()
// sb.reverse()
```

Классы-обертки (wrappers)

ИТМО

- Примитивные типы
  - int, long, byte, short
  - float, double, char, boolean
- Меньше памяти и быстрее
- Нет методов
- Не являются объектами

- Обертки
  - Integer, Long, Byte, Short
  - Float, Double, Character, Boolean
- Больше памяти и медленнее
- Много полезных методов
- Являются объектами

Автоупаковка/автораспаковка (auto(un)boxing)

ИТМО

```
int primitive = 42;
Integer wrapper;

wrapper = primitive;                    // автоупаковка
wrapper = Integer.valueOf(primitive); // как работает на самом деле

primitive = wrapper + 1;                // автораспаковка и сложение
primitive = wrapper.intValue() + 1;   // как работает на самом деле
```

Использование оберток - магия сравнения

ИТМО

```
Integer int1 = 42;
Integer int2 = 42;

System.out.println(int1 == int2);        // true
```

Использование оберток - магия сравнения

ИТМО

```
Integer int1 = 42;
Integer int2 = 42;

System.out.println(int1 == int2);        // true


int1 = 451;
int2 = 451;

System.out.println(int1 == int2);        // false
```

Использование оберток - магия сравнения

ИТМО

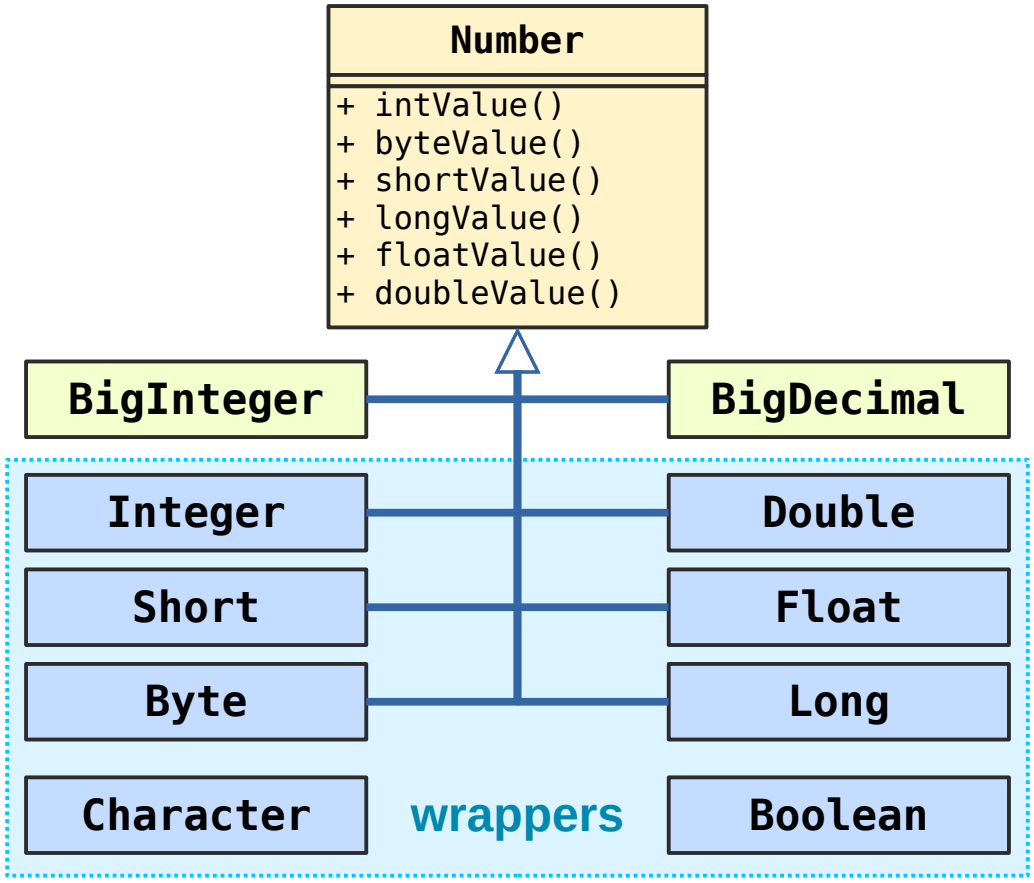
```
Integer int1 = 42;
Integer int2 = 42;

System.out.println(int1 == int2);         // true
System.out.println(int1.equals(int2));    // true

int1 = 451;
int2 = 451;

System.out.println(int1 == int2);         // false
System.out.println(int1.equals(int2));    // true
```

Классы BigInteger и BigDecimal

ИТМО

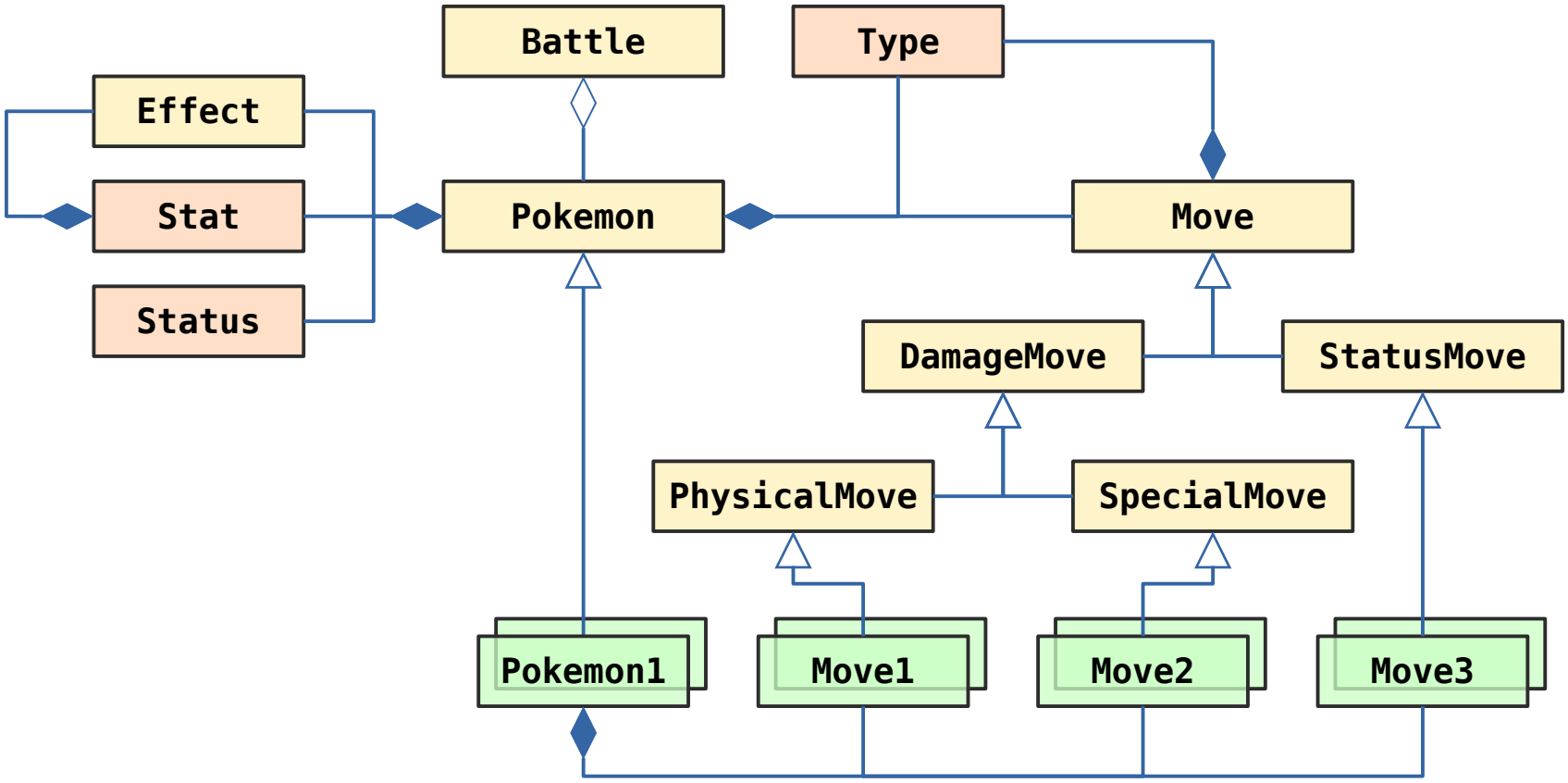
- BigInteger - целое с произвольной точностью
- BigDecimal - десятичная дробь с произвольной точностью
- Конструкторы принимают String
- Методы для операций, как для int и double, и другие
- import java.math.*;

Базовый класс Number и обертки



Лаба 2 - покемоны

ИТМО



Лаба 2 - покемоны

ИТМО

- Класс Move
  - для своих атак нужно написать наследников
  - метод attack(Pokemon attacker, Pokemon defender)
    - Этот метод автоматически вызывается во время сражения
    - Внутри него вызываются методы для расчета урона, а потом методы для применения урона к покемонам
    - Сам метод трогать не нужно
    - В документации написано, какие методы он вызывает.

Лаба 2 - покемоны

ИТМО

- Класс Move
  - атаки бывают 3 типов (посмотреть какой должен быть тип)
    - PhysicalMove - физическая атака (наносит физический урон)
    - SpecialMove - специальная атака (наносит специальный урон)
    - StatusMove - меняет состояние покемонов
  - отличаются наличием урона (StatusMove без него) или тем, какие атака и защита используются для расчета (физические или специальные)

Лаба 2 - покемоны

ИТМО

- Класс Move
  - остальные методы
    - либо оставить как есть (если устраивает)
    - либо переопределить в соответствии с вариантом
  - обычно переопределяются:
    - checkAccuracy() - если успех атаки не стандартный
    - calcRamnomDamage() - если не стандартная вероятность урона
    - applyOppDamage/applySelfDamage - если не стандартный урон
    - applyOppEffects/applySelfEffects - когда есть спецэффекты

Лаба 2 - покемоны

ИТМО

- Класс Pokemon
  - все методы - финальные, переопределять не надо
  - нужно сделать наследника - своего покемона
    - задать тип - setType
    - задать характеристики (HP, атаку, защиту, ...) - setStats
    - добавить атаки - addMove
    - передать в конструктор имя и уровень

Лаба 2 - покемоны

ИТМО

- Класс Effect
  - эффекты, изменяющие характеристики покемона
    - например, снижение атаки на 1 в течение 3 ходов с вероятностью 50%
    - `new Effect().chance(0.5).turns(3).stat(Stat.ATTACK, -1);`
  - эффекты накладываются
    - методом setCondition(Effect) у покемона
    - методами класса Effect, которым передается покемон

Лаба 2 - покемоны

ИТМО

- Перечисление Stat
  - статы покемона (HP, атака, защита, скорость, точность, ...)
- Перечисление Status
  - состояние покемона (сон, заморозка, паралич, яд, огонь)
- Перечисление Type
  - тип покемона (достаточно задать нужные)

Лаба 2 - покемоны

ИТМО

- Класс Battle
  - Распределить покемонов по командам
  - Запустить сражение
- Начать с базовых покемонов с базовыми атаками
  - Как заработает - постепенно добавлять функционал
  - Читать документацию по классам и методам

Лаба 2 - подключение внешнего .jar

ИТМО

- Как должно работать
  - Компилятор и JVM должны иметь возможность найти класс и при компиляции и при запуске
  - Список каталогов и jar-архивов, где искать классы, задается:
    - опцией -cp или -classpath или --class-path (через `:` или `;`)

```
javac -cp lib/pokemon.jar MyLab2.java
java -cp lib/pokemon.jar:my.jar MyLab2
```

    - в поле Class-Path: в манифесте .jar-архива (через пробел)

```
Class-Path: lib/pokemon.jar
Main-Class: MyLab2
```

Вопросы?