Объектно-ориентированное программирование

Лекция 4: принципы ООП и параметризованные типы

НГУ, ФИТ, А.А. Шадрина

28.09.23

Сегодня про

- Ключевые принципы ООП: абстракция, инкапсуляция, наследование
- Приведение типов “вверх” и “вниз”
- Отношения между классами
- Аккуратное использование наследования
- Параметризованные типы

С прошлых лекций

- Что такое объект?
- Что такое класс?
- Что такое тип данных?
- Какие типы данных бывают в Java?

Структура класса в Java

```
class Car {
    private int modelYear;
    private String modelName;

    public Car(int modelYear, String modelName) {
        this.modelYear = modelYear;
        this.modelName = modelName;
    }

    public int getModelYear() {
        return this.modelYear;
    }

    @Override
    public String toString() {
        return modelName + " " + modelYear;
    }
}
```

Структура класса в Java

```
class Car {
    private int modelYear;
    private String modelName;

    // ...

    public static void main(String[] args) {
        var car1 = new Car(2023, "Lada Granta");
        var car2 = new Car(2022, "Moskvitch 3");
        var car3 = new Car(2006, "Toyota RAV4");

        System.out.println(car3);
    }
}
```

Часть 1: Принципы ООП

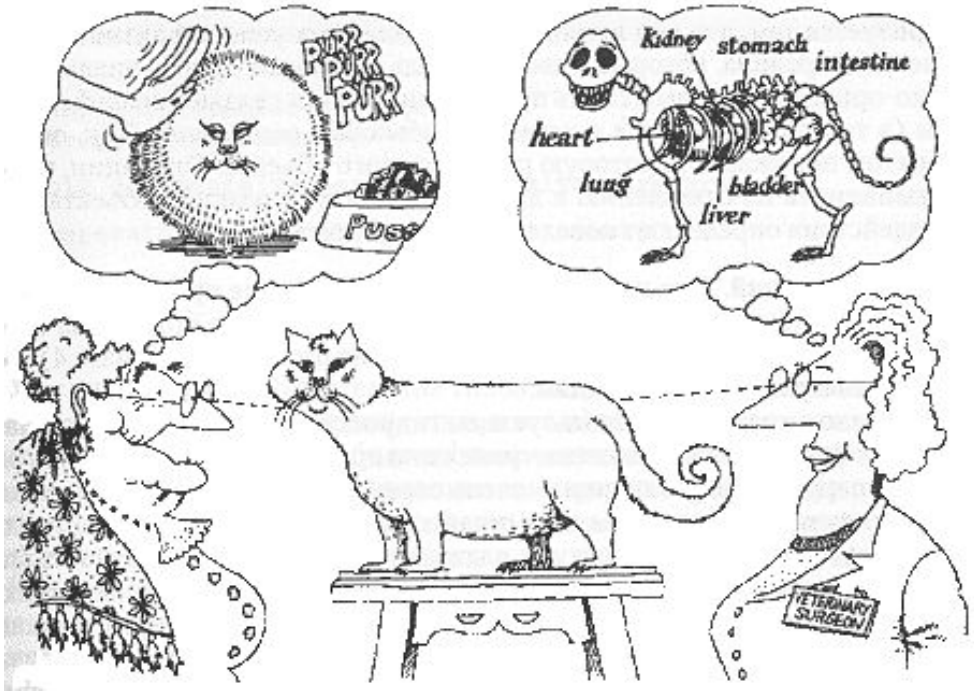
Принципы ООП

1. Абстракция
2. Инкапсуляция
3. Наследование
4. Полиморфизм
5. Типизация
6. Параллелизм
7. Сохраняемость

Абстракция

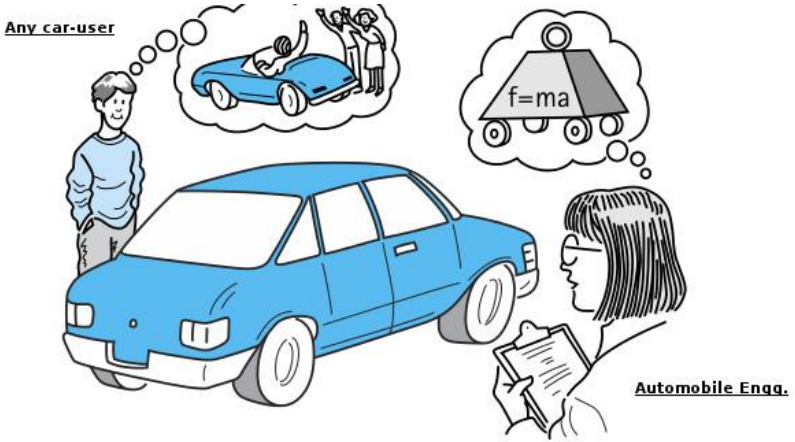
Абстракция выделяет **существенные характеристики** некоторого объекта ... и четко определяет его концептуальные границы **с точки зрения наблюдателя.**

Гради Буч



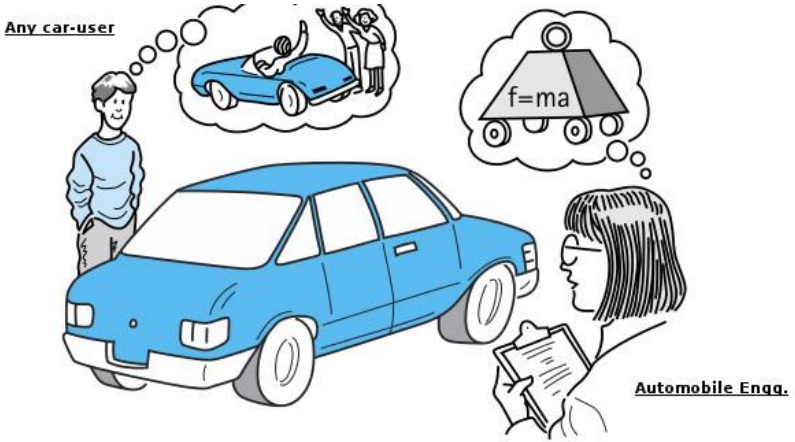
Абстракция

```
class Car {

    // ...

    public void start() {
        // ...
    }

    public void stop() {
        // ...
    }

    public int getPetrolValue() {
        // ...
    }
```



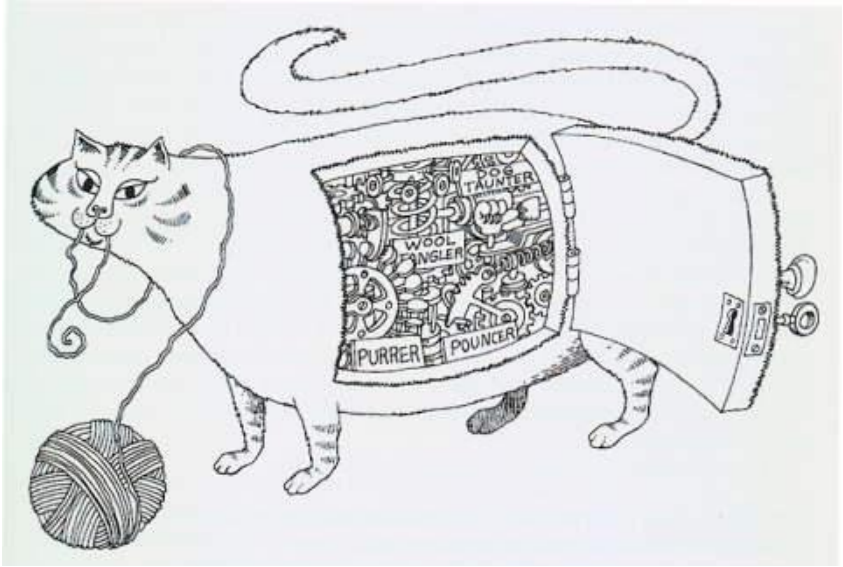
Абстракция

```
class Car {

    // ...

    public String getModelName() {
        // ...
    }

    public int getModelYear() {
        // ...
    }

    public void modifyPrice(int price) {
        // ...
    }
```



Инкапсуляция

Процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы **изолировать контрактные обязательства абстракции от их реализации**



Гради Буч

Инкапсуляция

```
class Car {
   private static int PETROL_MIN_ACCEPTABLE_VALUE = //...

   private int petrolValue;
   private boolean petrolShotageIndicator = false;

   public void start() {
      if (petrolValue < PETROL_MIN_ACCEPTABLE_VALUE) {
         turnOnPetrolShotageIndicator();
      }
      startEngine();
      // ...
   }

   private void turnOnPetrolShotageIndicator() {
      this.petrolShotageIndicator = true;
   }
```

Средства абстракции и инкапсуляции в Java

- Модификаторы доступа (`public`, `private`, package-private, `protected`)
- Интерфейсы (ключевое слово `interface`)
- Абстрактные классы (модификатор `abstract`)

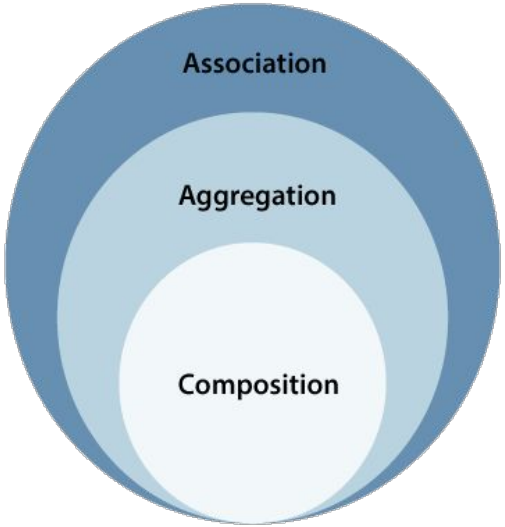
Модификаторы доступа

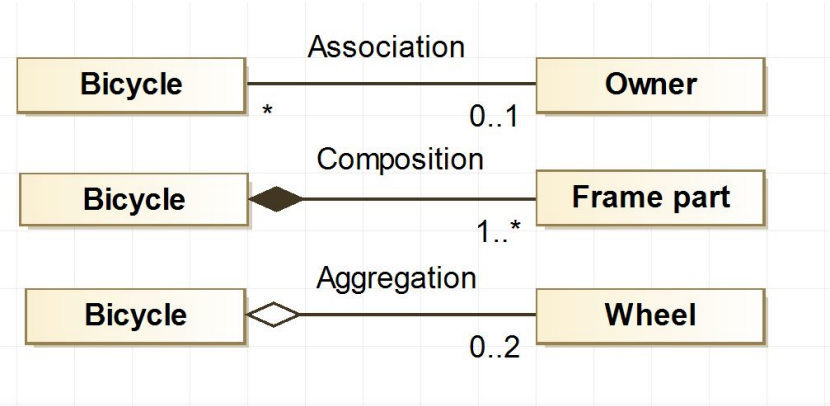
Модификаторы доступа задают видимость:

- `public` — для всех
- `private` — внутри класса
- package-private — внутри пакета
- `protected` — внутри пакета и в наследниках

Отношения между классами

1. Ассоциация
2. Зависимость
3. Агрегация (часть-целое)
4. Композиция (часть-целое + время жизни)
5. Наследование





Отношения между классами

```
public class Car {
    Engine engine;
    Passenger[] passengers;
    Wheel[] wheels;

    void changeWheel(CarJack carJack) {
        // ...
    }
}
```

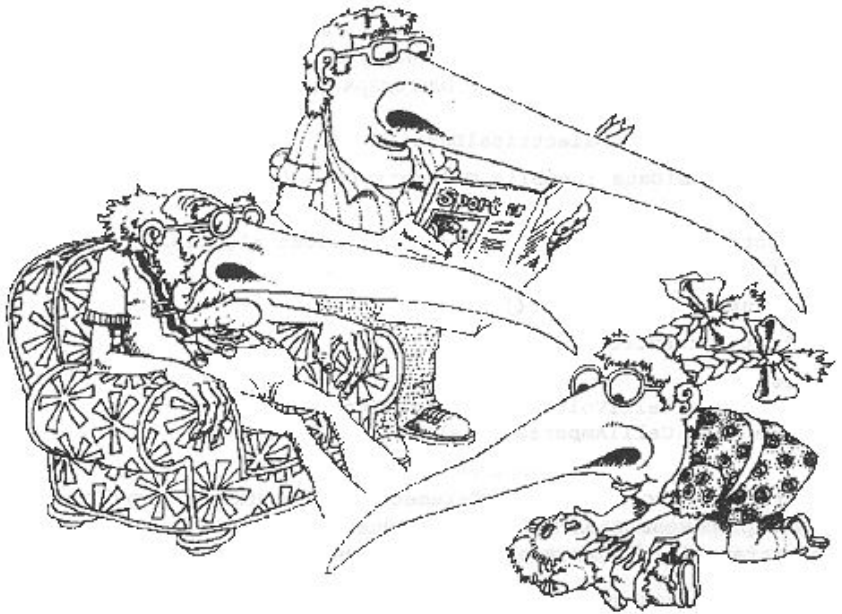
Отношения между классами

```
public class Car {
    Engine engine;           // Composition
    Passenger[] passengers; // Association
    Wheel[] wheels;          // Aggregation

    void changeWheel(CarJack carJack) { // Dependency
        // ...
    }
}
```

Наследование

Наследование создает такую **иерархию абстракций**, в которой подклассы заимствуют строение и функциональность от одного или нескольких суперклассов.

Иерархия – это упорядочение абстракций путем расположения их по уровням.



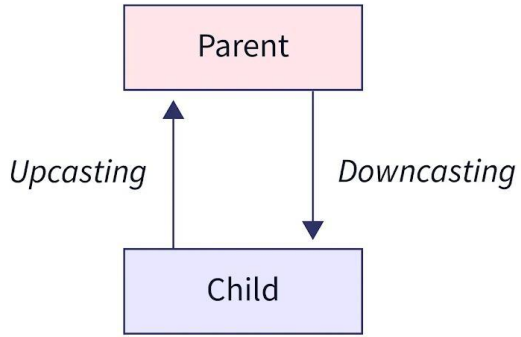
Наследование: пример

```java
class Pet {
    protected String name;
    private int age;

    Pet(String name, int age) {
        this.name = name;
        this.age = age;
    }

    protected void makeSound() {}
}
```

```java
class Cat extends Pet {
    Cat(String name, int age) {
        super(name, age);
    }

    @Override
    void makeSound() {
        System.out.println(super.name + "Meow");
    }
}
```

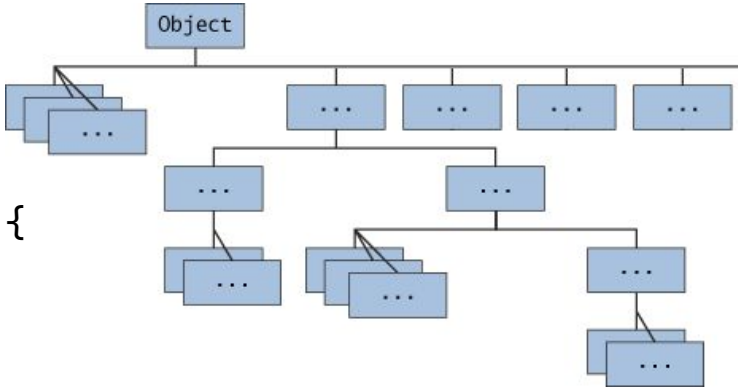
Наследование: upcast и downcast

```
Cat cat1 = new Cat("Murka", 5);
Cat cat2 = new Cat("Vasiliy", 15);
Dog dog1 = new Dog("Yasha", 3);

Pet pet = cat1;       // Upcast
pet.makeSound();      // Calls makeSound() of Cat!
Cat catCE = pet;      // CE, required type is Cat, Pet provided
Cat cat = (Cat) pet; // Downcast
Dog dog = (Dog) pet; // OK, but ClassCastException
```



Наследование: java.util.Object

```
public class Object {
   public native int hashCode();

   public boolean equals(java.lang.Object obj) {
       return (this == obj);
   }

   public String toString() {
       return getClass().getName() + "@"
                  + Integer.toHexString(hashCode());
   }

   // wait(), notify(), notifyAll()…
```



Наследование: LSP

Liskov substitution principle (LSP) — принцип подстановки Барбары Лисков:

*Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, **не зная об этом.***

*Наследующий класс должен **дополнять, а не замещать** поведение базового класса.*

Наследование: нарушение LSP

```
class Rectangle {
   private int width;
   private int height;

   public int getWidth() {
       return width;
   }

   public void setWidth(int width) {
       this.width = width;
   }

   public int getHeight() { … }

   public void setHeight(int height) { … }

   public int perimeter() {
       return 2 * height + 2 * width;
   }
}
```

```
class Square extends Rectangle {
   @Override
   public void setHeight(int height) {
       super.setHeight(height);
       super.setWidth(height);
   }

   @Override
   public void setWidth(int width) {
       super.setWidth(width);
       super.setHeight(width);
   }
}
```

Наследование: нарушение LSP

```
public void checkPerimeter(Rectangle rectangle) {
    rectangle.setHeight(5);
    rectangle.setWidth(7);

    int result = rectangle.perimeter();

    System.out.println(24 == result);
}
```

Composition over inheritance

```
class StringStorage {
   private String[] internalStorage;
   private int i;

   StringStorage() {
       internalStorage = new String[SIZE];
       i = 0;
   }

   public void addString(String s) {
       internalStorage[i++] = s;
   }

   public void addStrings(String[] ss) {
       for (String s : ss) {
           addString(s);
       }
   }

   public void deleteString(int index) {...}
}
```

Composition over inheritance

```
class StringStorage {
   private String[] internalStorage;
   private int i;

   StringStorage() {
       internalStorage = new String[SIZE];
       i = 0;
   }

   public void addString(String s) {
       internalStorage[i++] = s;
   }

   public void addStrings(String[] ss) {
       for (String s : ss) {
           addString(s);
       }
       i += ss.length;
   }

   public void deleteString(int index) {...}
}
```

```
class SpecialStringStorage extends StringStorage {
   int countOfAdds = 0;

   @Override
   public void addString(String s) {
       super.addString(s);
       countOfAdds++;
   }

   @Override
   public void addStrings(String[] ss) {
       super.addStrings(ss);
       countOfAdds += ss.length;
   }
}
```

Composition over inheritance

```
class Main {
   public static void main(String[] args) {
       var stringStorage = new SpecialStringStorage();
       stringStorage.addString("abc");
       stringStorage.addStrings(new String[] { "a", "b", "c" });

       System.out.println(stringStorage.countOfAdds);
   }
}
```

Composition over inheritance

```
class Main {
    public static void main(String[] args) {
        var stringStorage = new SpecialStringStorage();
        stringStorage.addString("abc");
        stringStorage.addStrings(new String[] { "a", "b", "c" });

        System.out.println(stringStorage.countOfAdds); // 7

    }
}
```

Composition over inheritance

Выводы:

- Наследование — самая сильная связь, которая может нарушить инкапсуляцию.
- Дочерний класс заимствует все недостатки API родителя.
- Предпочитайте композицию.

Принципы ООП

1. Абстракция
2. Инкапсуляция
3. Наследование
4. Полиморфизм
5. Типизация
6. Параллелизм
7. Сохраняемость

Резюмируя

- Абстракция выделяет существенные характеристики с точки зрения наблюдателя.
- Инкапсуляция скрывает детали внутреннего устройства.
- Существуют разные типы отношений между классами: ассоциация, зависимость, агрегация, композиция, наследование.
- Наследование создает иерархию абстракций.
- Наследованием требует внимательного использования, так как может нарушить инкапсуляцию.
- Следует предпочесть композицию наследованию, когда это резонно.

Часть 2: Параметризованные типы

Параметризованные типы: зачем?

Задача:

Реализовать контейнер **Box** для хранения значения, которое может иметь разный ссылочный тип.

Параметризованные типы: класс Box

```
public class Box {
    private Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

Параметризованные типы: класс Box

```
Box box = new Box();

box.setValue(42);
System.out.println("Value is " + box.getValue());

box.setValue("abc");
System.out.println("Value is " + box.getValue());

String takenFromBox = (String) box.getValue();

// ...

Integer takenFromBoxAgain = (Integer) box.getValue(); // RE!
```

Параметризованные типы: класс Box<T>

```
public class Box<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Параметризованные типы: класс Box<T>

```
Box<String> box = new Box<String>();

box.setValue(42); // CE

box.setValue("abc");
System.out.println("Value is " + box.getValue());

String takenFromBox = box.getValue(); // No cast

// ...

Integer takenFromBoxAgain = (Integer) box.getValue(); // CE
```

Параметризованные типы (generics)

- Возможность переиспользовать код для разных типов данных
- Более строгая проверка типов, вместо ClassCastException ошибка компиляции
- Явная спецификация типового параметра(ов)
- Не нужно лишнее приведение типа

```
var listOfStrings1 = new ArrayList<String>();
ArrayList<String> listOfStrings2 = new ArrayList<>();
```

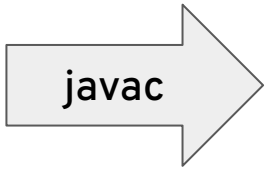
Параметризованные методы

```
static <T> void putInBoxIfEmpty(T value, Box<T> box) {
    if (box.getValue() == null) {
        box.setValue(value);
    }
}

class Box<T> {
    static <E> void putInBoxIfEmpty(E value, Box<E> box) {
        if (box.getValue() == null) {
            box.setValue(value);
        }
    }
    // ...
```

Стирание типов (type erasure)

Компилятор стирает информацию о типе, заменяя все параметры типом `Object`.

```java
public class Box<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```




```java
public class Box {
    private Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

Границы типовых параметров (bounds)

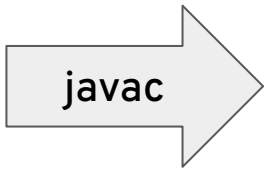
Типовые параметры могут иметь верхние (upper bound) и нижние (lower bound) границы.

```
public class Box<T extends Number> {
   private T value;

   public T getValue() {
       return value;
   }

   public void setValue(T value) {
       this.value = value;
   }
}
// ...

final var numberBox = new Box<Integer>();
final var stringBox = new Box<String>(); // CE
```

Стирание типов (type erasure)

Компилятор стирает информацию о типе, заменяя все параметры без ограничений (unbounded) типом `Object`, а параметры с границами (bounded) — на эти границы.

```java
public class Box<T extends Number> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```




```java
public class Box {
    private Number value;

    public Number getValue() {
        return value;
    }

    public void setValue(Number value) {
        this.value = value;
    }
}
```

Ограничения использования

- В качестве типового аргумента можно использовать только ссылочные типы

```
Box<int> box = new Box<int>(); // CE
```

- Статическое поле не может иметь тип типового параметра

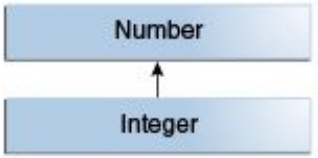
```
class A<T> {
   static T t; // CE
```

- Нельзя инстанцировать типовой параметр и массив типовых параметров

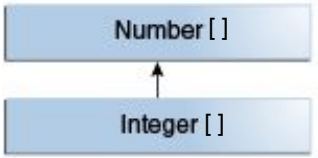
```
T t = new T();        // CE
T array = new T[10]; // CE
```

Вариантность: массивы

Массивы в Java ковариантны.



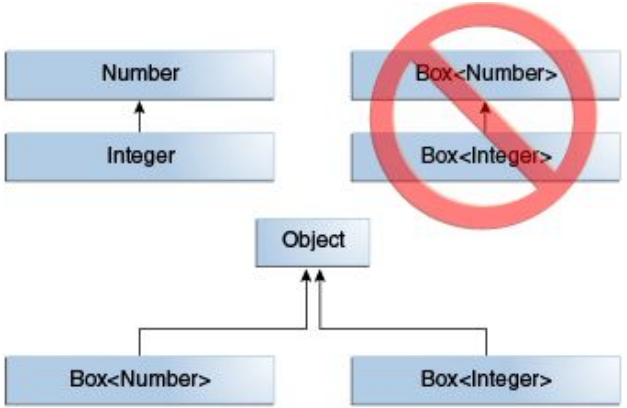

```
Number[] numbers = new Number[3];
numbers[0] = 42;   // java.lang.Integer
numbers[1] = 42.2; // java.lang.Double
numbers[2] = 33L;  // java.lang.Long

numbers = new Integer[3];
numbers[0] = 42.2; // RE: ArrayStoreException
```



Вариантность: параметризованные типы

Параметризованные типы в Java **инварианты**.

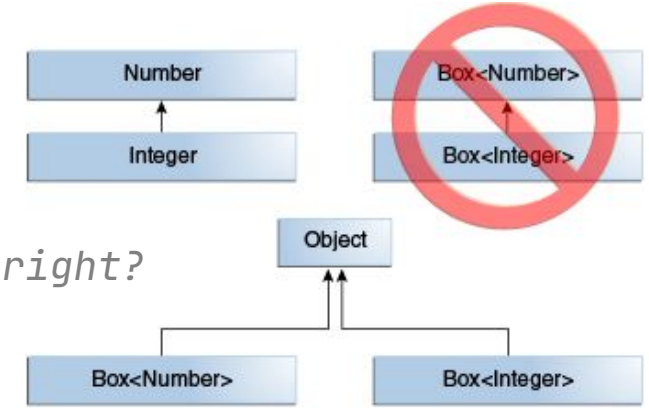
```
Box<Number> numberBox = new Box<Number>();
numberBox.setValue(42);
numberBox.setValue(42.2);

numberBox = new Box<Integer>(); // CE
```



Вариантность: параметризованные типы

Параметризованные типы в Java **инварианты**.

```
ArrayList<Dog> dogs = new ArrayList<Dog>();
ArrayList<Pet> animals = dogs;
animals.add(new Cat());
Dog dog = dogs.get(0); // This should be safe, right?
```



Вопросы?