

# НАСЛЕДОВАНИЕ И РАСШИРЯЕМОСТЬ СИСТЕМ. ПРОБЛЕМЫ НАСЛЕДОВАНИЯ




Александра  
Пшеборовская



# Александра Пшеборовская

QA Automation Lead

 [Александра Пшеборовская](#)



# План занятия

1. [Задача](#)
2. [Наследование](#)
3. [Тип переменной и тип объекта](#)
4. [Object](#)
5. [Полиморфизм и Override](#)
6. [Ограничения наследования](#)
7. [Итоги](#)



# ЗАДАЧА

# ЗАДАЧА

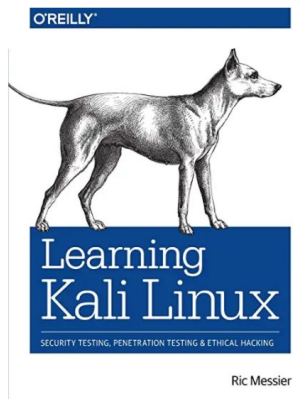
На прошлой лекции мы разобрали пример с корзиной покупок. На этой лекции нас будут интересовать уже сами товары:



ComputerGear Linux Polo Shirt Tux  
Computer Golf Geek Officially Licensed

★★★★☆ ~ 60

\$29<sup>99</sup>



Learning Kali Linux: Security Testing,  
Penetration Testing, and Ethical Hacking  
by Ric Messier

★★★★☆ ~ 23

Paperback

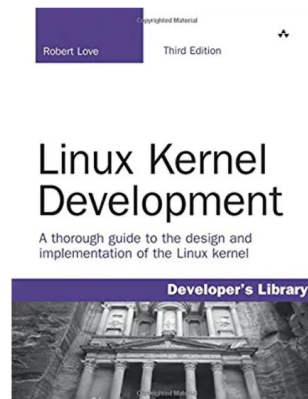
\$38<sup>49</sup> ~~\$49.99~~

Only 10 left in stock (more on the way).

More Buying Choices

\$25.55 (30 used & new offers)

Other format: [Kindle](#)



Linux Kernel Development  
by Robert Love

★★★★☆ ~ 99

Paperback

\$31<sup>49</sup> ~~\$49.99~~

More Buying Choices

\$20.00 (58 used & new offers)

Other format: [Kindle](#)

Результаты поиска по фразе «Linux» с сайта Amazon.com



# ЗАДАЧА

Наша задача — понять, как организовать систему, которая может хранить товары разных типов.



ComputerGear

## ComputerGear Linux Polo Shirt Tux Computer Golf Geek Officially Licensed

★★★★★ 60 ratings

Price: **\$24.99 - \$34.99**

Fit: **As expected (84%)**

Size:

Select

[Size Chart](#)

Color: Black

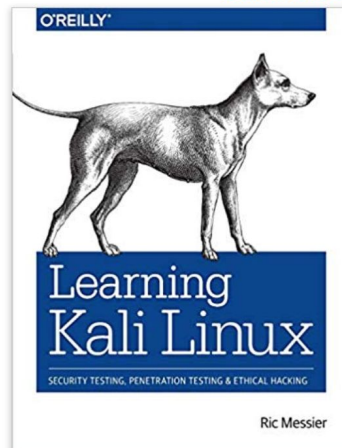


## Learning Kali Linux: Security Testing, Penetration Testing, and Ethical Hacking 1st Edition

by [Ric Messier](#) (Author)

★★★★★ 23 ratings

[Look inside](#)



ISBN-13: 978-1492028697

ISBN-10: 9781492028697

[Why is ISBN important?](#)

Kindle



**Paperback**

**\$29.53 - \$38.49**

Other Sellers

See all 2 versions

☐ Buy used

**\$29.53**

☒ Buy new

**\$38.49**

**Only 10 left in stock (more on the way).**

List Price: ~~\$49.99~~

Save: \$11.50 (23%)

**17 New from \$38.49**

+ \$8.98 shipping

This item ships to **Austria**. **Want it Monday, March 30?**  
Order within **6 hrs 59 mins** and choose **AmazonGlobal**  
**Priority Shipping** at checkout. [Learn more](#)

[Deliver to Austria](#)

Qty: 1



Add to Cart



Buy Now

# ЗАДАЧА

Так как у футболок и книг совершенно разные свойства, у нас есть всего два варианта:

1. Сделать один большой класс, в который поместить все возможные свойства со всех типов продаваемых товаров.
2. Сделать по отдельному классу на каждый тип товаров.

Но оба решения обладают недостатками:

- в первом решении - очень много избыточных полей
- во втором решении - "не получится" поместить объекты разных классов в один массив\*

Примечание\*: чуть позже мы узнаем, что на самом деле - получится.





# НАСЛЕДОВАНИЕ



## ОБЩИЕ ХАРАКТЕРИСТИКИ

*Вопрос к аудитории: какие общие характеристики вы можете выделить у книги и футболки?*

# PRODUCT

Если мы выделим общие характеристики, то получится такой класс:

```
public class Product {  
    private int id;  
    private String name;  
    private int price;  
  
    // + constructors/getters/setters*  
}
```

Примечание\*: мы сначала рассмотрим всё без Lombok. Мы используем сгенерированные с помощью IDEA геттеры и сеттеры (без логики), конструкторы без параметров и с параметрами.

# НАСЛЕДОВАНИЕ

**Наследование** — механизм, позволяющий строить новые классы, расширяя уже существующие. При этом эти новые классы получают все поля и методы родительских классов.

Например, мы можем создать новый класс `Book`, расширяя класс `Product`, добавив в него новые поля:

```
public class Book extends Product {  
    private String author;  
    private int pages;  
    private int publishedYear;  
}
```

расширяем класс `Product`



# ТЕРМИНЫ

1. Базовый класс, супер-класс, родительский класс, родитель — тот, от кого наследуемся (стоит справа от слова **extends**).
2. Подкласс, производный класс, дочерний класс, ребёнок, унаследованный класс — тот, кто наследуется (стоит слева от слова **extends**).

# КАРТИНКИ

Нарисуем схематично, как это будет организовано:

```
public class Book extends Product {
```

```
    private int id;  
    private String name;  
    private int price;
```

```
    private String author;  
    private int pages;  
    private int publishedYear;
```

```
}
```

```
public class Product {
```

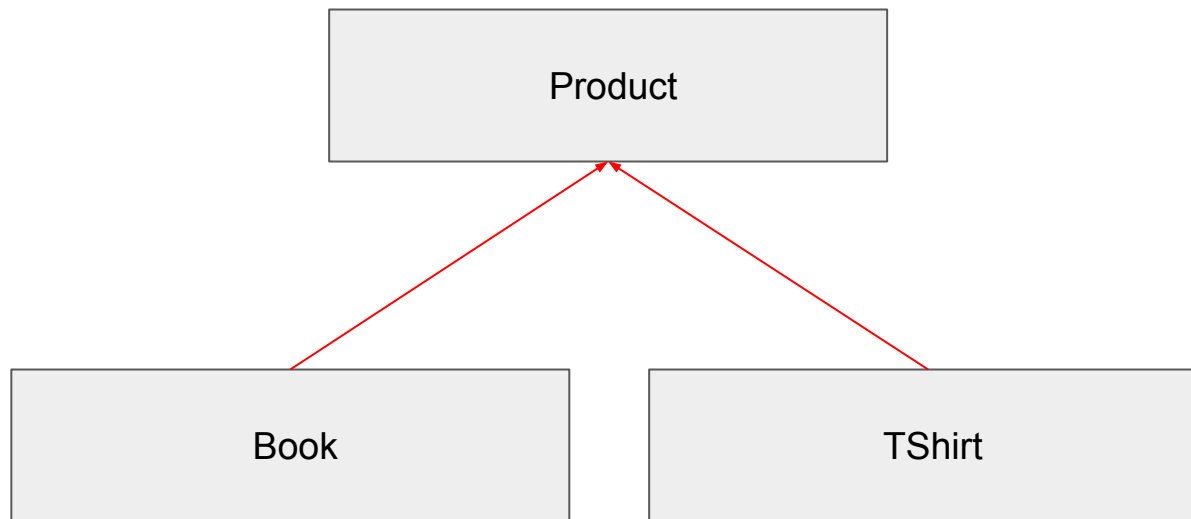
```
    private int id;  
    private String name;  
    private int price;
```

```
}
```

Представьте, что поля и методы (кроме конструкторов), просто скопировались в дочерний класс.

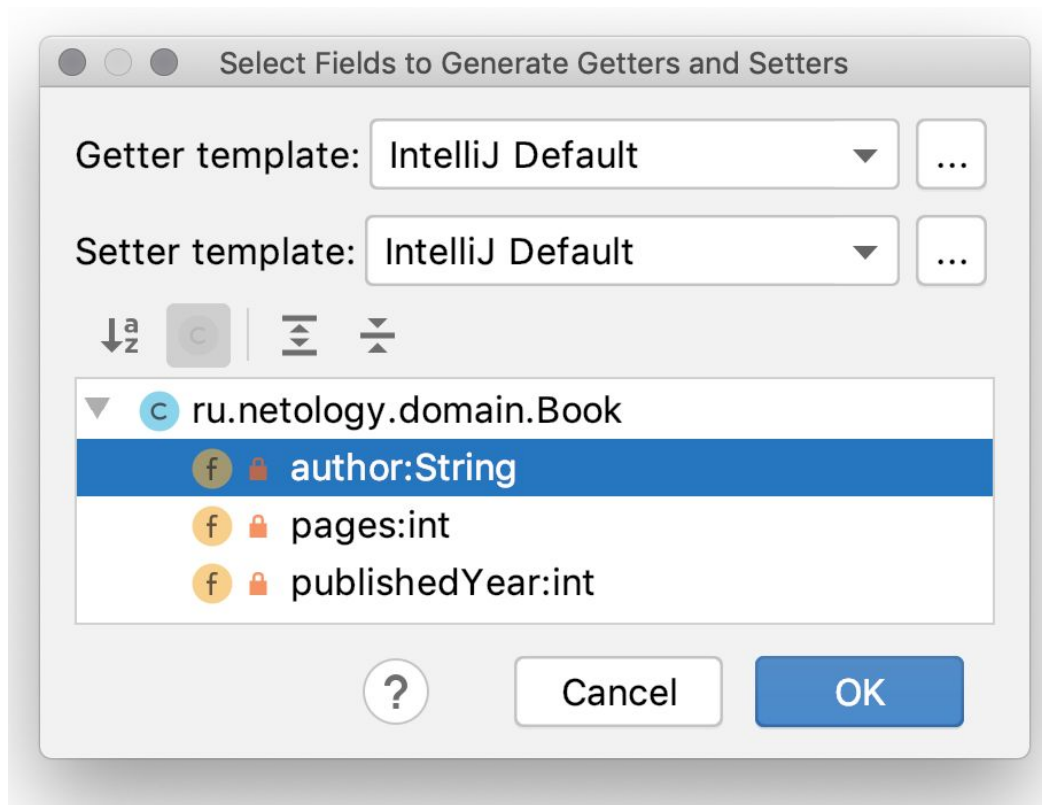
Это верная аналогия, с одним исключением: если поля в родительском классе были **private**, то внутри дочернего класса **доступа к ним нет**.

# ИЕРАРХИЯ



# GETTERS/SETTERS

При генерации в IDEA нам предложат только поля дочернего класса (т.к. поля родительского класса не видны):





# СМОТРИМ, ЧТО ЭТО НАМ ДАЛО

```
class BookTest {  
    @Test  
    public void shouldHaveAllFieldsAndMethodFromSuperClass() {  
        Book book = new Book();  
        book.  
    }  
}
```

|   |                                     |        |
|---|-------------------------------------|--------|
| m | getAuthor()                         | String |
| m | getPages()                          | int    |
| m | getPublishedYear()                  | int    |
| m | setAuthor(String author)            | void   |
| m | setPages(int pages)                 | void   |
| m | setPublishedYear(int publishedYear) | void   |
| m | getId()                             | int    |
| m | getName()                           | String |
| m | getPrice()                          | int    |
| m | setId(int id)                       | void   |
| m | setName(String name)                | void   |
| m | setPrice(int price)                 | void   |

Press ^. to choose the selected (or first... [Next Tip](#)

← методы самого класса

← методы родителя

# НАСЛЕДОВАНИЕ

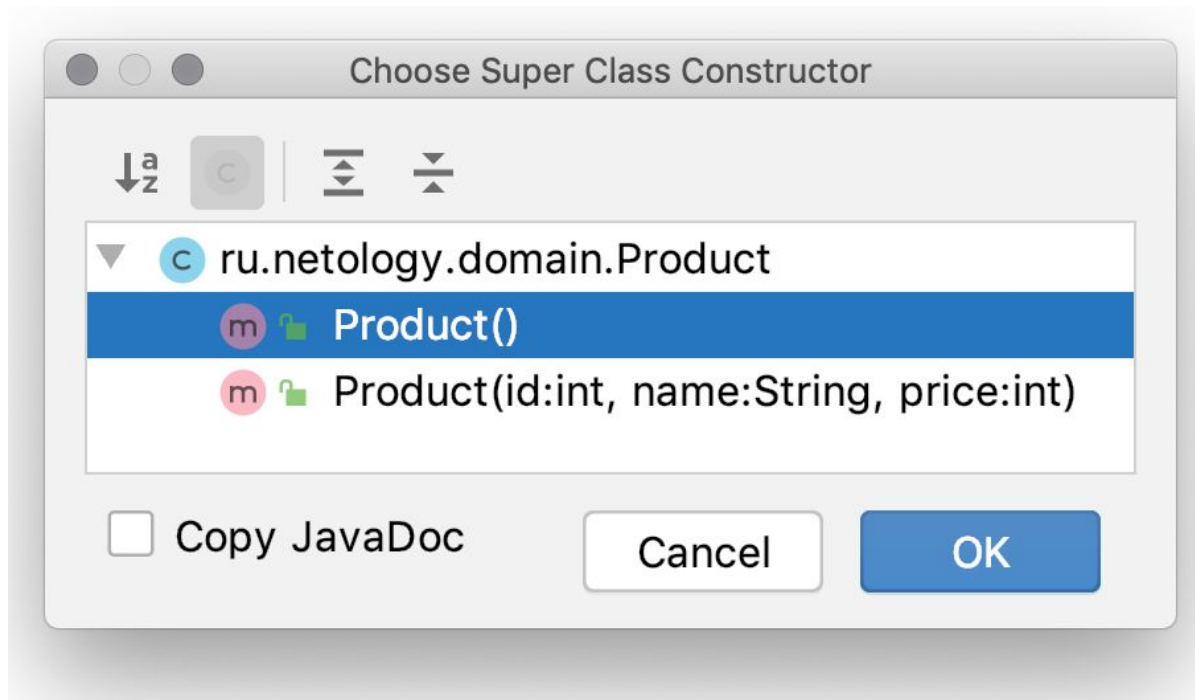
Таким образом, мы получили возможность переиспользовать методы (и поля) родительского класса в дочернем, не дублируя код:

```
public class TShirt extends Product {  
    private String color;  
    private String size;  
}
```

Но что с конструкторами и инициализацией?

# КОНСТРУКТОР

Попробуем с помощью IDEA (без Lombok) сгенерировать конструктор для нашего дочернего класса:



# КОНСТРУКТОРЫ

```
public Book() {  
    super();  
}  
  
public Book(int id, String name, int price, String author, int pages, int publishedYear) {  
    super(id, name, price);  
    this.author = author;  
    this.pages = pages;  
    this.publishedYear = publishedYear;  
}
```

С конструкторами всё немного сложнее: каждый конструктор\* должен вызывать конструктор родительского класса (для того, чтобы поля в родительском классе тоже были проинициализированы).

Примечание\*: на самом деле у него есть выбор: не вызывать явно (тогда используется дефолтный), вызывать явно или вызывать другой конструктор (но не родителя).

# SUPER

```
public Book() {  
    super();  
}  
  
public Book(int id, String name, int price, String author, int pages, int publishedYear) {  
    super(id, name, price);  
    this.author = author;  
    this.pages = pages;  
    this.publishedYear = publishedYear;  
}
```

Ключевое слово **super** отвечает за две функции:

1. Вызов конструктора родителя
2. Обращение к полям и методам родителя (если их видимость выше, чем **private**)



# НАСЛЕДОВАНИЕ

**Q:** Мы научились «выносить» общие характеристики в базовый класс, чтобы не дублировать их в дочерних классах. Но что это дало в контексте решения задачи?

**A:** Наследование позволяет нам использовать объекты дочерних классов везде, где требуются объекты родительских классов.

Давайте посмотрим на примере.

# НАСЛЕДОВАНИЕ

```
public class ProductRepository {  
    private Product[] items = new Product[0];  
  
    public void save(Product item) {  
        ...  
    }  
  
    public Product[] findAll() {  
        ...  
    }  
  
    public void removeById(int id) {  
        ...  
    }  
}
```

# НАСЛЕДОВАНИЕ

```
class ProductRepositoryTest {  
    private ProductRepository repository = new ProductRepository();  
    private Book coreJava = new Book();  
  
    @Test  
    public void shouldSaveOneItem() {  
        repository.save(coreJava);  
  
        Product[] expected = new Product[]{coreJava};  
        Product[] actual = repository.findAll();  
        assertEquals(expected, actual);  
    }  
}
```

кладём Book





## КЛЮЧЕВОЕ

Наследование — это **отношения типа «является»**.

Таким образом, мы говорим, что книга является продуктом (товаром), поэтому везде, где требуется товар, мы можем использовать книгу (книга — это и есть товар).

То же самое будет с футболкой.

# НАСЛЕДОВАНИЕ

**Q:** То есть теперь мы можем класть объект любого класса-наследника `Product`? А если будем доставать, то что придёт — `Product` или объект этого типа (например, `Book`)?

**A:** Это хороший вопрос. Давайте в нашей репозитории напомним метод, который будет искать продукт по его идентификатору:

```
public Product findById(int id) {  
    for (Product item : items) {  
        if (item.get)  
    }  
}
```



The image shows an IntelliJ IDEA autocomplete popup. It lists methods with a red circle icon containing an 'm' for methods. The methods are: `getId()` (returns `int`), `getName()` (returns `String`), `getPrice()` (returns `int`), and `getClass()` (returns `Class<? extends Product>`). At the bottom, it says "Press ↵ to insert, → to replace" and "Next Tip" with a vertical ellipsis icon.

| Method                  | Return Type                                 |
|-------------------------|---|
| <code>getId()</code>    | <code>int</code>                            |
| <code>getName()</code>  | <code>String</code>                         |
| <code>getPrice()</code> | <code>int</code>                            |
| <code>getClass()</code> | <code>Class&lt;? extends Product&gt;</code> |

← только методы `Product`



# ТИП ПЕРЕМЕННОЙ И ТИП ОБЪЕКТА



# ТИПЫ

Пример из жизни: в самолёте мы все пассажиры (кроме обслуживающего персонала), и когда мы смотрим на всех, как на пассажиров, то у всех них есть базовый набор методов и полей (например, номер места, возможность заказать напиток или вызвать стюардессу).

Но когда в самолёте кому-то становится плохо, то спрашивают «Есть ли доктор?», и если есть, то магическим образом из обычного пассажира этот человек превращается в доктора с методом «лечить».

# ТИПЫ

В Java тип переменной (аргумента или поля) **может отличаться** от типа объекта, который на самом деле хранится:

```
@Test
public void shouldCastToBaseClass() {
    Product product = new Book();
    product.~
}
```

тип переменной

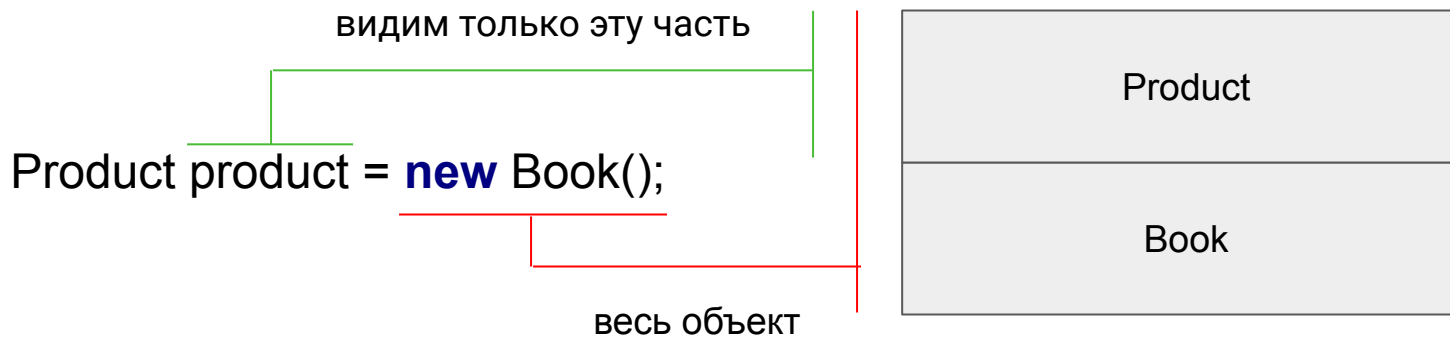
тип объекта

|   |                      |         |
|---|----------------------|---------|
| m | getId()              | int     |
| m | getName()            | String  |
| m | getPrice()           | int     |
| m | setId(int id)        | void    |
| m | setName(String name) | void    |
| m | setPrice(int price)  | void    |
| m | equals(Object obj)   | boolean |
| m | hashCode()           | int     |
| m | toString()           | String  |

# ТИПЫ

Важно: тип переменной определяет то, какие поля и методы мы видим (видим только те, что определены в типе переменной).

Но сам объект может обладать гораздо большим набором полей и методов:



## Q & A

Q: Как мы можем достать «остальную» часть?

A: Для этого существует механизм cast'инга (приведения типов):

@Test

```
public void shouldCastFromBaseClass() {
```

```
    Product product = new Book();
```

```
    if (product instanceof Book) {
```

```
        Book book = (Book) product;
```

```
        book.
```

|   |                                     |        |
|---|-------------------------------------|--------|
| m | getAuthor()                         | String |
| m | getPages()                          | int    |
| m | getPublishedYear()                  | int    |
| m | setAuthor(String author)            | void   |
| m | setPages(int pages)                 | void   |
| m | setPublishedYear(int publishedYear) | void   |

# INSTANCEOF

Оператор проверяет «безопасность» приведения типов, т.к. может возникнуть ситуация, при которой мы случайно попытаемся привести не к тому типу:

@Test

```
public void shouldNotCastToDifferentClass() {  
    Product product = new Book();  
    TShirt shirt = (TShirt) product;  
}
```

```
java.lang.ClassCastException: class ru.netology.domain.Book  
    cannot be cast to class ru.netology.domain.TShirt
```





# INSTANCEOF

Чаще всего, обилие **instanceof** в коде считается признаком «плохого кода», но вы должны знать, что он существует и для чего он нужен.



# OBJECT



# OBJECT

В Java существует специальный класс **Object**, который представляет из себя вершину иерархии классов.

Т.е. любой класс в Java (который явно не наследуется от другого класса), на самом деле наследуется от **Object**.

# OBJECT

При этом **extends** Object обычно не пишут:



```
public class Product extends Object {
```

```
    private int id;
```

```
    private String
```

```
    private int pri
```

Class 'Product' explicitly extends 'java.lang.Object'

[Remove redundant 'extends Object'](#)

[More actions...](#)



# ОБЪЕКТ

В этом классе содержится несколько ключевых методов, которые нам интересны:

```
public native int hashCode(); // хэш-код для хранения в структурах данных

public boolean equals(Object obj) { // проверка объектов на равенство
    return (this == obj);
}

public String toString() { // вывод объекта в строковом представлении
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

## Q & A:

Q: Что такое «проверка объектов на равенство»? Мы же говорили, что объекты сравниваются по ссылкам?

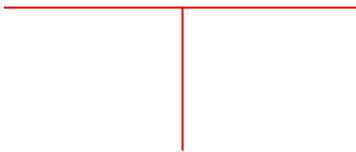
A: Совершенно верно, но стандартная библиотека Java и почти все инструменты используют метод equals:

```
public boolean equals(Object obj) {  
    return (this == obj); // сравниваем ссылки  
}
```

# ASSERT EQUALS



Если долго путешествовать по библиотеке JUnit, выясняя, как на самом деле работает *assertEquals*, то рано или поздно мы докопаемся до этого кода:

```
static boolean objectsAreEqual(Object obj1, Object obj2) {  
    if (obj1 == null) {  
        return (obj2 == null);  
    }  
    return obj1.equals(obj2);  
}
```



на первом объекте вызывается equals  
и передаётся второй объект

# ASSERT EQUALS

```
7   class ProductTest {  
8      @Test  
9   public void shouldUseEquals() {  
10      Product first = new Product( id: 1, name: "Java I", price: 1000);  
11      Product second = new Product( id: 1, name: "Java I", price: 1000);  
12      assertEquals(first, second);  
13  }  
14 }
```

org.opentest4j.AssertionFailedError:

Expected :ru.netology.domain.Product@5c30a9b0

Actual :ru.netology.domain.Product@1ddf84b8

[<Click to see difference>](#)



---

## Q & A

**Q:** Хорошо, но *assertEquals* всё равно сравнивает ссылки?

**A:** Да, но в Java существует механизм переопределения методов и полиморфизм.



# **ПОЛИМОРФИЗМ И OVERRIDE**

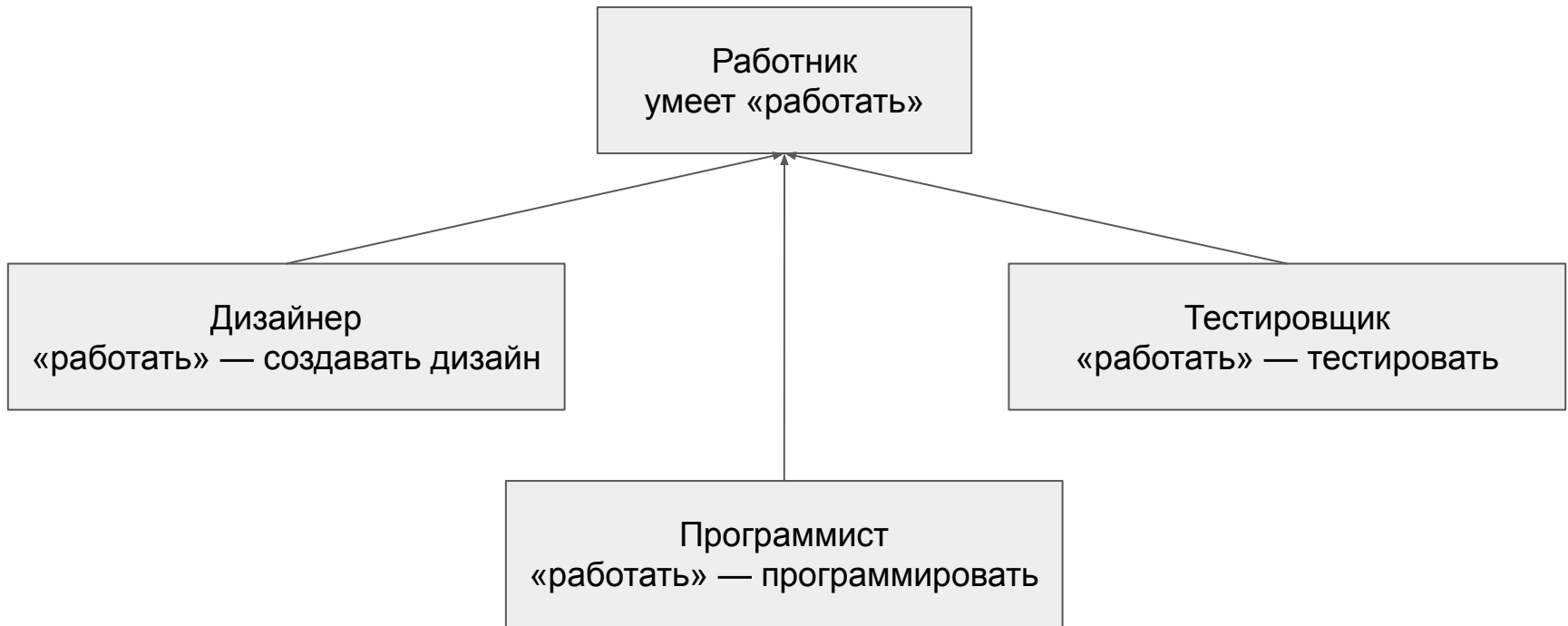


# ПОЛИМОРФИЗМ

**Полиморфизм** — механизм, при котором дочерние классы могут переопределять поведение методов родительского класса (при этом сигнатуры должны совпадать).

Доступные методы определяются типом переменной (аргумента или поля), а **вызываемый метод определяется типом объекта**.

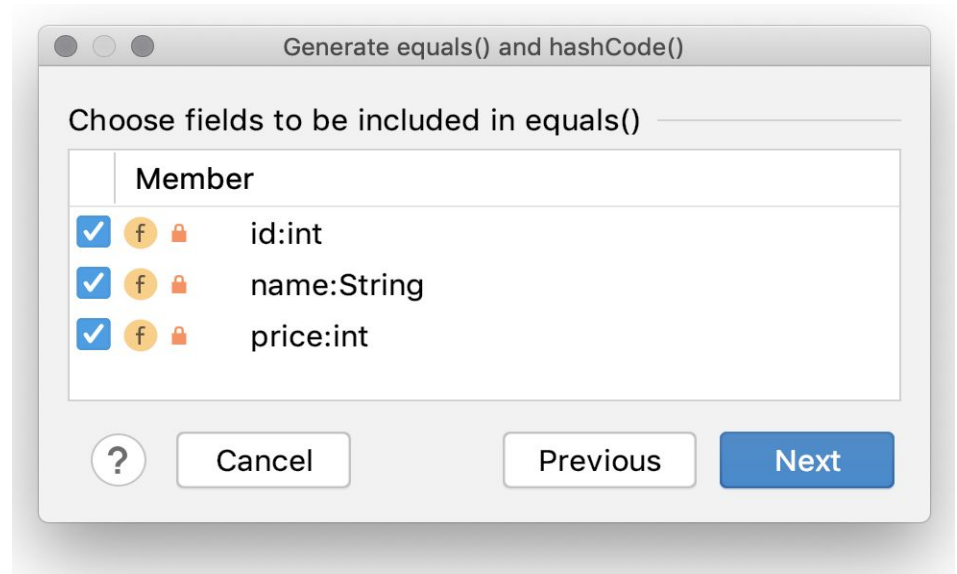
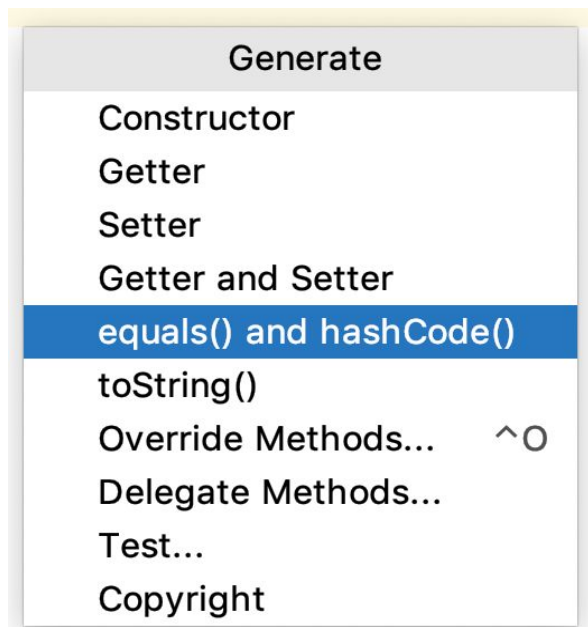
# ПРИМЕР ИЗ ПРОШЛЫХ ЛЕКЦИЙ



При этом все они являются «работниками» (или сотрудниками).

# В JAVA

В Java мы можем переопределить этот метод так, чтобы использовался именно наш метод, а не метод, определённый в классе **Object**:



---

## B JAVA

@Override

```
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (o == null || getClass() != o.getClass()) return false;  
    Product product = (Product) o;  
    return id == product.id &&  
        price == product.price &&  
        Objects.equals(name, product.name);  
}
```

@Override

```
public int hashCode() {  
    return Objects.hash(id, name, price);  
}
```

# КЛЮЧЕВЫЕ МОМЕНТЫ

**if (this == o) return true;** — early exit (если ссылки совпадают, то ничего больше не проверяем)

**if (o == null || getClass() != o.getClass()) return false;** — проверка на то, что объекты относятся к одному классу

**Product product = (Product) o;** — приведение типов (cast'инг)

**return id == product.id &&**

**price == product.price &&**

**Objects.equals(name, product.name)** — проверка на «равенство» полей

## **&&, ||, !**

Напоминаем, про три логических оператора:

1. **&&** — true тогда и только тогда, когда оба выражения (и справа, и слева — true)
2. **||** — false тогда и только тогда, когда оба выражения (и справа, и слева — false)
3. **!** — из true делает false, из false — true (унарный оператор)



# ТЕСТ

Теперь наш тест зелёный, поскольку используется именно наша версия equals:

```
class ProductTest {  
    @Test  
    public void shouldUseEquals() {  
        Product first = new Product( id: 1, name: "Java I", price: 1000);  
        Product second = new Product( id: 1, name: "Java I", price: 1000);  
        assertEquals(first, second);  
    }  
}
```

# ВАЖНО

Вы можете представлять этот механизм следующим образом:

1. Java ищет метод `equals` в классе, из которого был создан ваш объект. Если находит его, то использует.
2. Если не находит, то идёт в родительский класс и ищет там, если находит, то использует.
3. Если не находит, продолжает выполнять п.2, пока не дойдёт до `Object` (а в `Object` этот метод всегда есть).

Продемонстрировать в дебаггере вызов `equals` в `Product`.

# ПЕРЕОПРЕДЕЛЕНИЕ EQUALS

Мы можем переопределить метод equals и в классе Book:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    if (!super.equals(o)) return false;
    Book book = (Book) o;
    return pages == book.pages &&
        publishedYear == book.publishedYear &&
        Objects.equals(author, book.author);
}
```

Обратите внимание, что вызывается equals родительского класса и таким образом, обеспечивается сравнение всех полей.



# EQUALS

Важно: equals — это соглашение, а не обязательство.

Вы переопределяете его тогда, когда хотите, чтобы два разных объекта в памяти считались эквивалентными при выполнении определённых условий.



## ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

В дополнительных материалах к сегодняшней лекции в репозитории с ДЗ будут прикреплены материалы с описанием:

1. Механизма работы ключевого слова `static`
2. Обзор механизма рефлексии
3. Информация об использовании Lombok с наследованием

Обязательно с ними ознакомьтесь!

## Q & A

**Q:** Что это за вид при печати объекта? Это его адрес в памяти?

`org.opentest4j.AssertionFailedError:`

`Expected :ru.netology.domain.Product@5c30a9b0` ←

`Actual :ru.netology.domain.Product@1ddf84b8` ←

[<Click to see difference>](#)

**A:** Нет, на самом деле это просто работа метода `toString`, который вызывается тогда, когда вы пытаетесь распечатать объект (продемонстрировать в дебаггере). Вы также можете его переопределить.

# ПОЛИМОРФИЗМ

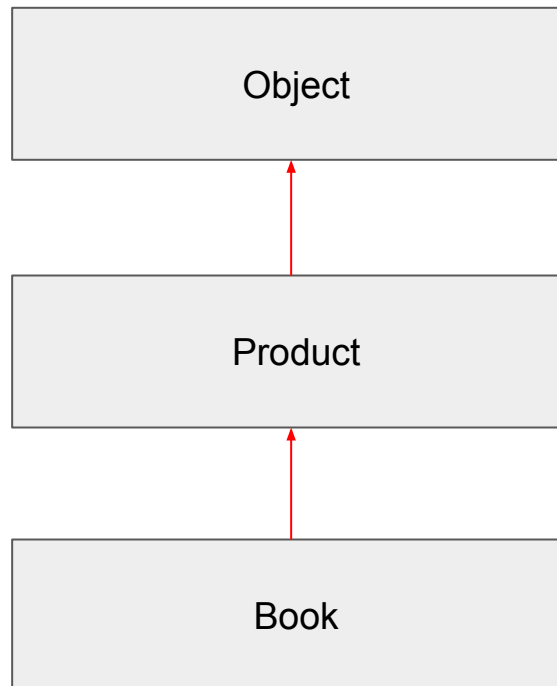
Полиморфизм — одна из ключевых концепций языка Java, позволяющая создавать нам расширяемые системы: дочерние классы переопределяют поведение родительских, но по-прежнему совместимы со всей системой типов.

Напоминаем: вызываемый метод определяется именно типом самого объекта, а не типом переменной:

```
@Test
public void shouldUseOverriddenMethod() {
    Product product = new Book();
    // Вопрос к аудитории: чей метод вызовется?
    product.toString();
}
```

# ПОЛИМОРФИЗМ

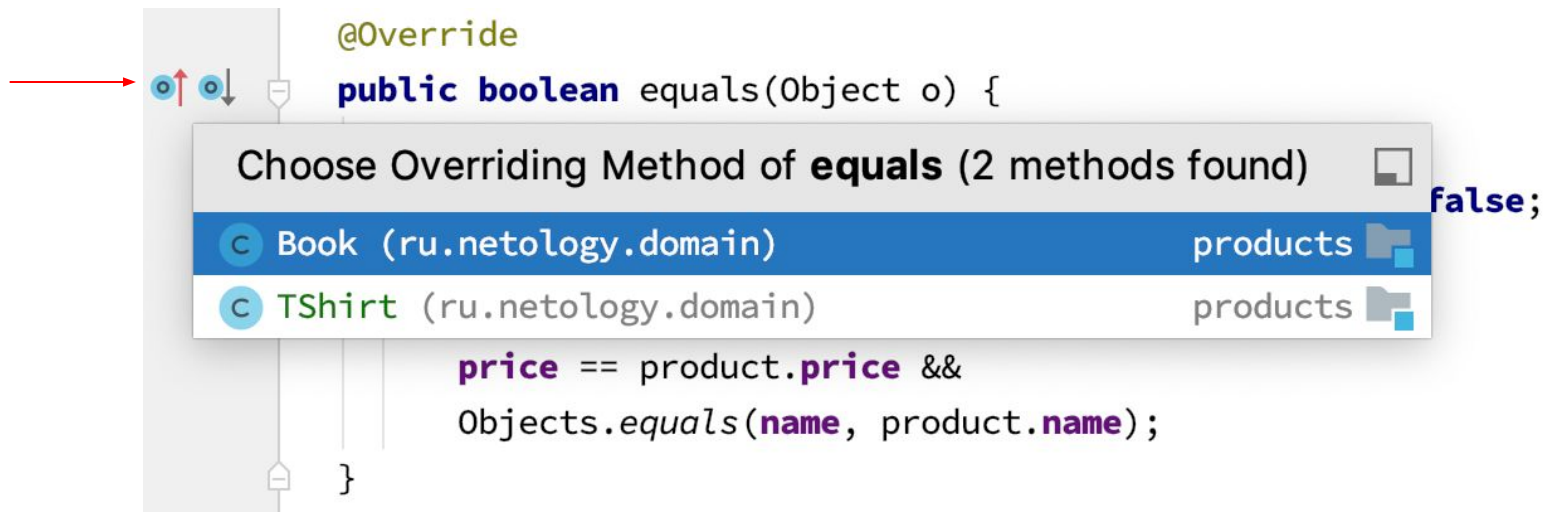
Вопрос на самом деле не обычный, всё зависит от того, в каком из этих классов переопределён toString (начиная снизу вверх):






# IDEA

IDEA позволяет вам увидеть, кто и чьи методы переопределяет с помощью специальных пиктограмм:





# **ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ**



## ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ

Отвлечёмся от текущей задачи и рассмотрим следующую:  
у нас есть МФУ (многофункциональное устройство), а также есть  
принтер и сканер.

*Вопрос к аудитории: как бы вы выстроили иерархию наследования?*



# ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ

Наследование — достаточно сильное ограничение для классов, в Java можно **наследоваться только от одного класса**.

МФУ, с одной стороны, является принтером (поскольку позволяет печатать), с другой стороны — сканером (поскольку позволяет сканировать).

# ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ

Но в Java мы не можем унаследоваться от двух классов, чтобы получить и метод `print`, и метод `scan`:

```
class Printer {  
    public void print() {}  
}
```

```
class Scanner {  
    public void scan() {}  
}
```

```
class MFU extends Printer, Scanner {  
}
```

Class cannot extend multiple classes

# ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ

Если подумать, то можно предположить, что МФУ — это всего лишь коробка для принтера и сканера (вспомните композицию):

```
class MFU {  
    private Printer printer;  
    private Scanner scanner;  
  
    public void print() {  
        printer.print();  
    }  
  
    public void scan() {  
        scanner.scan();  
    }  
}
```



## ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ

Но тогда мы теряем возможность использовать МФУ там, где нужен принтер или сканер (поскольку отношений типа «является» уже нет — композиция их не обеспечивает).

Мы вернемся к решению этой задачи на лекции про интерфейсы.



## ОГРАНИЧЕНИЯ НАСЛЕДОВАНИЯ

Пока же вы должны понимать, что унаследовавшись от какого-то класса, вы автоматически теряете возможность унаследоваться от другого (в Java запрещено множественное наследование).





# ИТОГИ



## ИТОГИ

Сегодня мы рассмотрели важную тему наследования и убедились, что наследование в Java используется повсеместно (любой класс, ни от кого явно не наследующийся, наследуется от класса `Object`).



## ИТОГИ

Тем не менее, наследование — очень жёсткая связь, поскольку позволяет встроить ваш класс только в одну ветку иерархии.



## ИТОГИ

В домашней работе, используя инструкции, вы доделаете реализацию системы, которую мы рассматривали на лекции, реализовав менеджер продукта в двух версиях.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаём в чате Slack!
- Задачи можно сдавать по частям.
- Зачёт по домашней работе проставляется после того, как приняты **все задачи**.



**Задавайте вопросы и напишите отзыв о лекции!**

**Александра Пшеборовская**

 [Александра Пшеборовская](#)