

INHERITANCE & POLYMORPHISM

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*

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

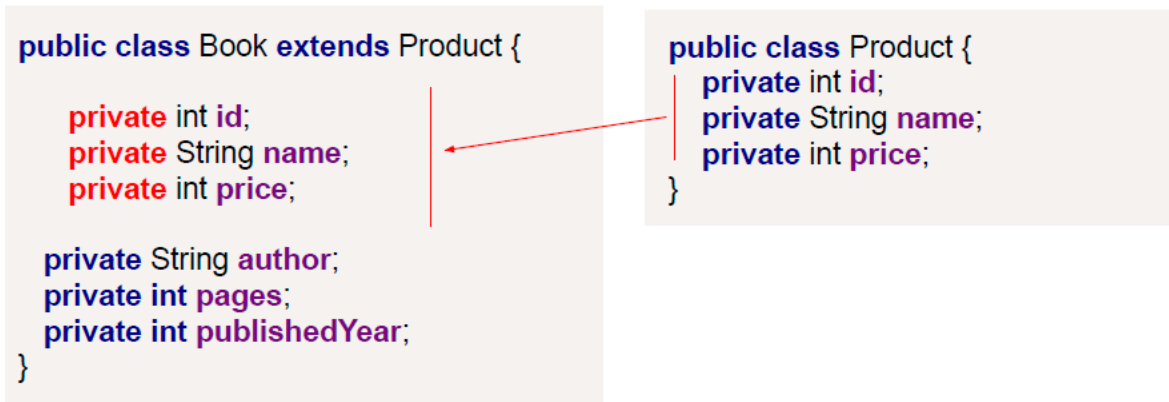
1. Базовый класс, супер-класс, родительский класс, родитель — тот, от кого наследуемся (стоит справа от слова **extends**).
2. Подкласс, производный класс, дочерний класс, ребёнок, унаследованный класс — тот, кто наследуется (стоит слева от слова **extends**).

При генерации Getters & Setters для дочернего классов в IDEA нам предложат только поля дочернего класса (т.к. поля родительского класса не видны):

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;  
}
```

Нарисуем схематично, как это будет организовано:



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**.

```
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

```

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).

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**)

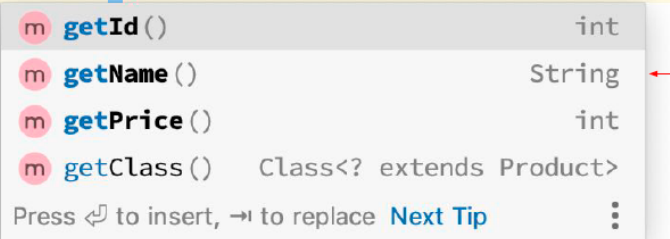
КЛЮЧЕВОЕ

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).

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 напишем метод, который будет искать продукт по его идентификатору:

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




только методы `Product`

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.
```

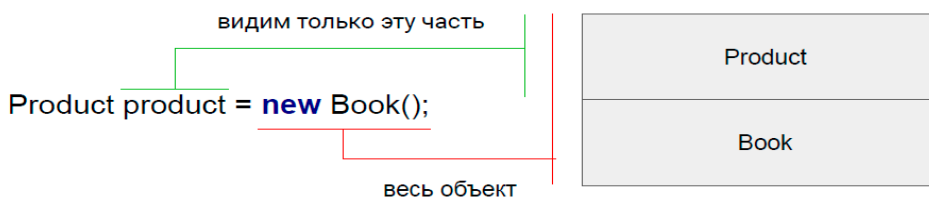


тип объекта

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: 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

INSTANCEOF

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

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

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**

При этом **extends** Object обычно не пишут

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

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

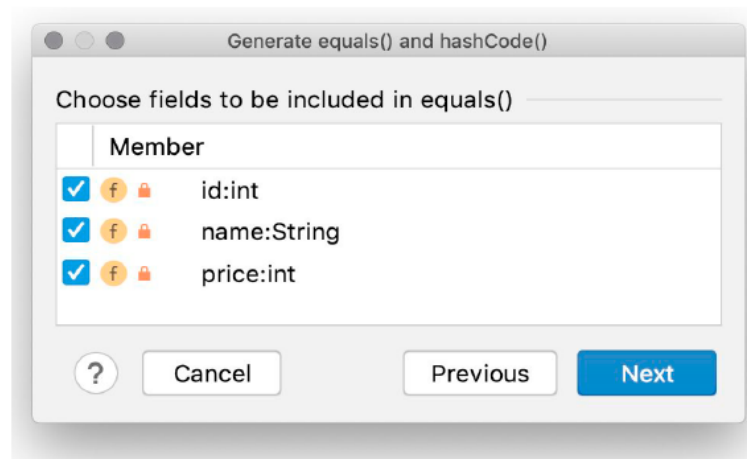
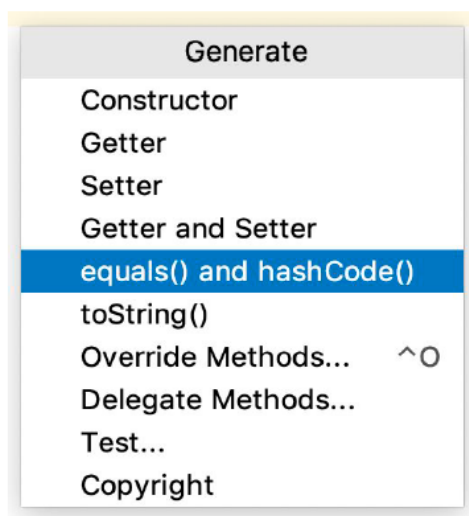
```
public boolean equals(Object obj) { // проверка объектов на равенство
    return (this == obj);
}
```

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

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**

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);  
}
```

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

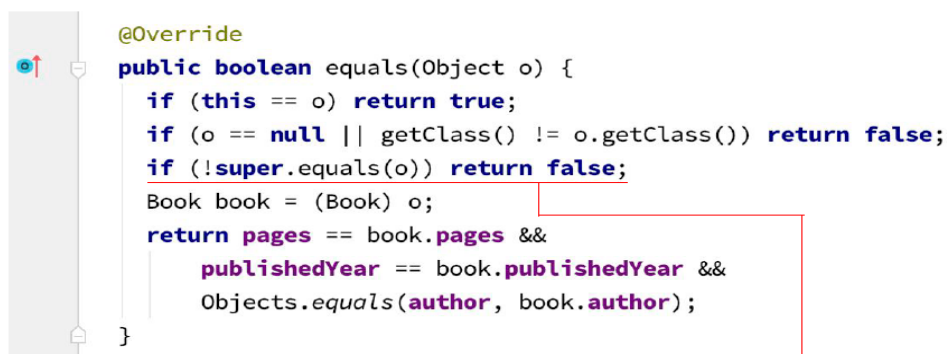
Напоминаем, про три логических оператора:

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)

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).

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.

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.

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 ←

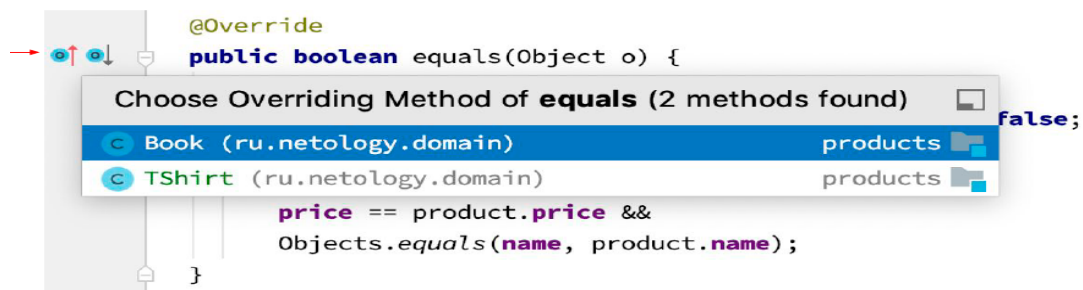
[<Click to see difference>](#)

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 — 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();
}
```

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



Унаследовавшись от какого-то класса, вы автоматически теряете возможность унаследоваться от другого (в Java запрещено множественное наследование).

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