



План занятия

1. Принципы ООП. Инкапсуляция
2. Что такое пакет и как его создать
3. Модификаторы доступа
4. Геттеры и сеттеры
5. Наследование
6. Вступаем в наследство
7. Переопределяем методы
8. Ключевое слово - super
9. Добавляем слово this в конструкторы



Принципы ООП

Принципы ООП

Java — объектно-ориентированный язык, и это значит, что в основе всех программ лежат объекты и классы. Под классом понимается шаблон или общее описание атрибутов и методов, а объект — это конкретный экземпляр. Из набора взаимодействующих между собой объектов и состоит любая программа в Java.





Принципы ООП

В методологии объектно-ориентированного программирования (ООП) есть четыре ключевых принципа:

- инкапсуляция
- наследование
- полиморфизм
- абстракция

Эти принципы отражают ключевые особенности написания кода на объектно-ориентированных языках — в том числе и на Java.



Инкапсуляция



В капсуле

Инкапсуляция (от англ. *encapsulation* и **от лат. *in capsula* — в капсуле) — это концепция, согласно которой данные и методы для работы с ними объединяются в «капсуле», например, внутри класса.



Инкапсуляция

Инкапсуляция важна по следующим причинам:

- Не нужно знать, как устроен внутренний процесс, достаточно внешнего интерфейса.



Инкапсуляция

Инкапсуляция предусматривает сокрытие внутреннего устройства объектов класса — взаимодействие с ними происходит при помощи внешнего интерфейса. К примеру, вы хотите приготовить капучино с помощью кофемашины. Вам не нужно в подробностях знать её устройство и механизм работы — достаточно поднести чашку и нажать пару кнопок.


```
public class CoffeeMachine {  
    private double volume = 300.0; // Объём чашки может быть "зашит" в программу  
    private boolean isAlmondMilk = true; // Машина может работать с разными видами молока  
  
    private void cleanMachine() {  
        System.out.println("Очистка машины..");  
    }  
  
    private void warmWater() {  
        System.out.println("Нагрев воды..");  
    }  
  
    private void grindGrain() {  
        System.out.println("Дробление зёрен..");  
    }  
  
    private void addMilk() {  
        System.out.println("Добавление молока..");  
    }  
  
    private void createFoam() {  
        System.out.println("Добавляем пену..");  
    }  
}
```

```
public String getCappuccino() {
    cleanMachine(); // Тут, например, дополнительно происходит очистка
    warmWater(); // Нагреваем воду в нагревателе
    grindGrain(); // Размельчаем зерна
    addMilk(); // А здесь мы дополнительно добавляем молоко
    createFoam(); // Создаём пенку
    return "Капучино!";
}

public String getLatte() {
    cleanMachine();
    warmWater();
    grindGrain();
    addMilk(); // В латте гораздо меньше пены, поэтому отдельный метод не вызываем
    return "Латте!";
}
```

Роль кнопок кофемашины, интерфейса, в коде выполняют методы:

```
public class Practice {  
    public static void main(String[] args) {  
        //Есть кофемашина:  
        CoffeeMachine coffee = new CoffeeMachine();  
  
        /*Пользователь заказывает капучино,  
        внутри кофемашины что-то происходит, и она готовит капучино*/  
        System.out.println(coffee.getCappuccino());  
  
        /*Пользователь заказывает латте,  
        внутри кофемашины что-то происходит, и она готовит латте*/  
        System.out.println(coffee.getLatte());  
    }  
}
```



Результат

Очистка машины..
Нагрев воды..
Дробление зёрен..
Добавление молока..
Добавляем пену..
Капучино!
Очистка машины..
Нагрев воды..
Дробление зёрен..
Добавление молока..
Латте!



Инкапсуляция

- Высокая скорость понимания кода.

Этот пункт напрямую следует из предыдущего. Если погружаться в реализацию каждого класса и метода, то потребуется много времени и ресурсов. Представьте, если бы каждый раз перед тем, как арендовать автомобиль в каршеринге, нужно было бы изучать, как работает карбюратор и двигатель конкретной модели. Вместо этого мы просто садимся за руль и едем.

То же самое при написании кода. Вспомните, насколько сложной была реализация методов в финансовом приложении. Однако благодаря понятным именам вам не требовалось снова и снова разбираться в алгоритмах их работы — достаточно было просто их вызвать.



Инкапсуляция

- Удобство работы с кодом.

Допустим, вы написали метод `getG()`, где для расчётов использовалось ускорение свободного падения на поверхности Земли с точностью до одного знака после запятой — 9,8. Этой точности оказалось недостаточно и понадобилось использовать значение 9,80665. Благодаря инкапсуляции вам достаточно поменять реализацию метода только в одном месте — исправлять везде в коде значение 9,8 на 9,80665 не придётся.



Инкапсуляция

- Минимизация ошибок и надёжное хранение данных.

Инкапсуляция предполагает ограничение видимости данных. Во многих случаях она работает как надёжная охранная система — с сейфами, прочными стенами и железными дверьми. Если бы методы одного класса могли бесконтрольно менять переменные в другом классе, то число ошибок выросло бы в разы. Благодаря инкапсуляции можно снизить число возможных побочных эффектов.



Реализация в коде

В коде инкапсуляция реализуется за счёт следующих инструментов:

- методов, включая **set**- и **get**- методы,
- пакетов,
- модификаторов доступа.

Пакеты структурируют классы в программе. Соккрытие данных достигается с помощью модификаторов доступа: **private**, **public**, **protected** и модификатора по умолчанию **default (package-private)** — каждый из них определяет свой уровень доступности. Методы позволяют реализовать интерфейс. С помощью **set**- и **get**- методов можно настроить доступ к данным.



Что такое пакет и как его создать



Что такое пакет и как его создать

До этого момента в заданиях уроков вы встречали не так много классов — от двух до четырёх. В реальных проектах их могут быть сотни или даже тысячи. Чтобы не запутаться, классы нужно систематизировать. Для этого в Java есть **пакеты**.



Что такое пакет и как его создать

Поскольку каждый класс в Java хранится в отдельном файле, то по сути пакет — это папка (*namespace* — с англ. «пространство имён») с файлами классов. Пакеты похожи на привычные папки на компьютере, где хранятся фотографии, фильмы или музыка. Точно так же, как внутри папки с фильмами могут быть отдельные папки с триллерами или романтическими комедиями, внутри пакета можно хранить не только классы, но и другие пакеты.



Что такое пакет и как его создать

Пакеты объединяют классы по смыслу и назначению. Это отражено в их названиях. Например, в пакете **billing** (англ. «счёт») будут классы, связанные с тратами, платежами или денежными переводами, в пакете **util** (от англ. *utility* — «польза») собраны классы-помощники. Названия пакетов в отличие от классов пишутся со строчной буквы.

Когда классы сгруппированы в пакеты — это сильно упрощает процесс навигации по коду в проекте. Благодаря тому, что у пакетов понятные названия, найти в них нужную информацию можно легко и быстро.





Импорт пакетов

В предыдущих уроках вы активно работали с пакетом `java.util` и такими его классами, как `ArrayList`, `Scanner`, `Random` и другими. Пакет `java.util` один из самых широко используемых — он входит в стандартную библиотеку Java и содержит большое число вспомогательных классов.



Что такое пакет и как его создать

С помощью пакетов можно избежать конфликтов при использовании классов с одинаковыми именами.

Представьте, что в программе есть класс **Printer**, который просто печатает в консоль какой-то текст, и есть класс **Printer** с полями, такими как модель, скорость печати, цвет текста, и методами — напечатать календарь или расписание и другими. Такие классы лучше добавить в разные пакеты и ограничить их видимость. Это поможет избежать ошибок при компиляции кода.





Модификаторы доступа



Модификаторы доступа

Когда нужно что-то скрыть или оградить от внешнего вторжения, то в повседневной жизни можно использовать разные ограничители — замок или забор, маску или чёрные очки, сейф или табличку с предупреждением. В Java эту роль играют **модификаторы доступа**. Они позволяют скрывать или, наоборот, делать общедоступными разные части программы: методы, поля и классы.



Модификаторы доступа

Всего в Java четыре модификатора. Для обозначения трёх из них в коде используются ключевые слова **public**, **private**, **protected**. Четвёртый не имеет обозначения — это модификатор по умолчанию **default** или **package-private**.



Что такое модификаторы доступа

- A. Абстракция для увеличения видимости переменных, классов и методов.
- B. Инструмент для изменения переменной или метода.
- C. Набор ключевых слов для обозначения видимости классов, полей и методов.
- D. Набор ключевых слов для создания новых объектов класса.



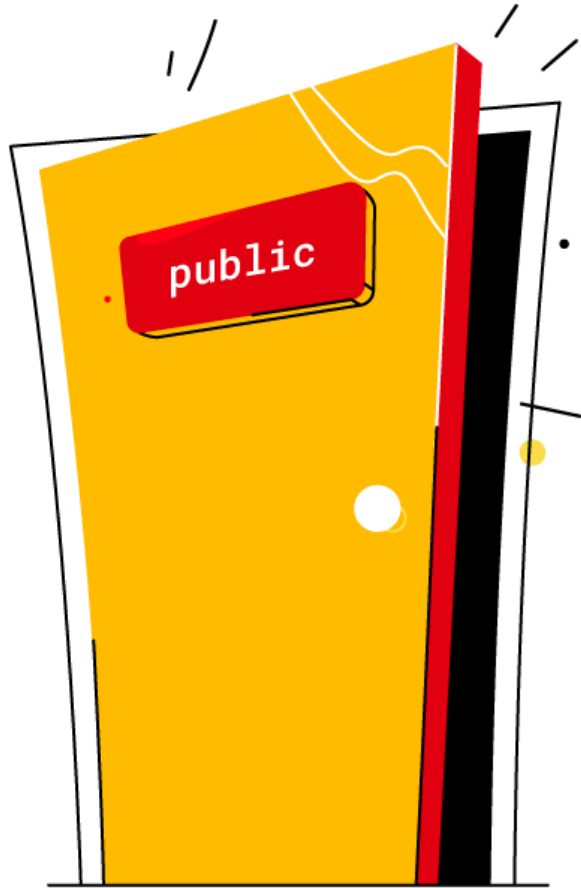
Ответ

A. Абстракция для увеличения видимости переменных, классов и методов.
При помощи модификаторов можно как увеличивать видимость, так и ограничивать.

B. Инструмент для изменения переменной или метода.
Модификаторы доступа ограничивают или открывают видимость, но не меняют содержание.

C. Набор ключевых слов для обозначения видимости классов, полей и методов.
Если же ключевое слово отсутствует — сработает модификатор по умолчанию.

D. Набор ключевых слов для создания новых объектов класса.
Модификаторы **public**, **private**, **protected** нужны для работы с видимостью данных, а не для того, чтобы создавать объекты.





public

При объявлении классов вы уже не раз встречали модификатор **public** (англ. «публичный»), но не знали, зачем он нужен. **public** означает, что помеченные им класс, поле или метод будут видны другим классам, в том числе в других пакетах. У **public** самая высокая область видимости.

К примеру, изучите код калькулятора, который считает количество рабочих часов в месяце с учётом, что один рабочий день длится восемь часов. Он хранится в классе **WorkCalculator**. Чтобы у классов из других пакетов был доступ к переменной **workingHours** (длительности рабочего дня) и методу **calculate()** (считает число отработанных часов в зависимости от количества дней) выбран модификатор **public**:

```
package com.practice.calculator; // пакет calculator

public class WorkCalculator {
    public int workingHours = 8; // теперь переменная доступна в любом классе

    public int calculate(int workDays) { // метод также доступен в любом классе и пакете
        return workDays * workingHours;
    }
}
```



```
package com.practice.calendar; // пакет calendar
// импортировали класс WorkCalculator из пакета calculator
import com.practice.calculator.WorkCalculator;

public class Main {
    public static void main(String[] args) {
        WorkCalculator calculator = new WorkCalculator();
        int daysJanuary = 15;
        // без препятствий вызвали метод calculate
        System.out.println("Рабочие часы за январь: " +
calculator.calculate(daysJanuary));
        // смогли обратиться к переменной workingHours
        System.out.println("Один день - " + calculator.workingHours + " часов.");
    }
}
```



Результат

Рабочие часы за январь: 120
Один день - 8 часов.



public

Модификатор **public** даёт классу **Main** из одного пакета доступ к классу **WorkCalculator** из другого, а также к методу **calculate()** и переменной **workingHours**.

Если в одном файле хранится несколько классов, то только один из них может иметь модификатор **public**. Имя файла при этом должно совпадать с именем класса с **public**. Другие классы должны иметь более низкий уровень доступа.



private

Противоположность **public** — модификатор **private** (англ. «приватный»). Он обеспечивает максимальную закрытость данных — полностью ограничивает доступ к помеченным им классам, полям и методам. Это означает, что их можно использовать только внутри их же класса. Из других классов или пакетов доступа к таким данным нет.

Изменим в классе **WorkCalculator** модификаторы у переменной и метода на **private**. Тогда поменять значение **workingHours** и «накрутить» число отработанных часов не получится:

```
package com.practice.calculator; // пакет calculator

public class WorkCalculator {
    private int workingHours = 8; // теперь переменная доступна только в этом классе

    private int calculate(int workDays) { //метод тоже доступен только здесь
        return workDays * workingHours; // обращаемся к переменной внутри её класса
    }
}
```

```
package com.practice.calendar; // пакет calendar  
// импортировали класс WorkCalculator из пакета calculator  
import com.practice.calculator.WorkCalculator;  
  
public class Main {  
    public static void main(String[] args) {  
        WorkCalculator calculator = new WorkCalculator();  
        // здесь произойдёт ошибка компиляции - нет доступа к методу  
        calculator.calculate(10);  
        // переменная workingHours не доступна  
        System.out.println("Рабочие часы: " + calculator.workingHours);  
    }  
}
```


Результат





```
Main.java:9:19  
java: calculate(int) has private access in com.practice.calculator.WorkCalculator
```

Если запустить класс **Main**, то в консоли появится сообщение **calculate(int) has private access in com.practice.calculator.WorkCalculator** — у метода **calculate(int)** установлен доступ **private** в классе **WorkCalculator**. При работе в среде разработки IDEA также появится подсказка, что метод и переменная имеют **private**-модификаторы и недоступны в **Main**.

```
package com.practice.calendar; // пакет calendar
// импортировали класс WorkCalculator из пакета calculator
import com.practice.calculator.WorkCalculator;




public class Main {
    public static void main(String[] args) {
        WorkCalculator calculator = new WorkCalculator();
        // здесь произойдет ошибка компиляции - нет доступа к методу
        calculator.calculate(10);
        // переменная workingHours недоступна
        System.out.println("Рабочие часы: " + calculator.workingHours);
    }
}
```

'workingHours' has private access in 'com.practice.calculator.WorkCalculator' 

Make 'WorkCalculator.workingHours' public   More actions...  

 com.practice.calculator.WorkCalculator

private int workingHours = 8

 testingStudents  



Задача

Исправьте ошибку доступа в классе **Practice**.

```
public class Practice {  
    public static void main(String[] args) {  
        Product product = new Product("Пирожок", 180.0);  
        System.out.println("Стоимость продукта - " +  
            product.getPrice() + " тг.");  
    }  
}  
  
public class Product {  
    private String name;  
    private double price;  
  
    public Product(String nameProduct, double priceProduct) {  
        name = nameProduct;  
        price = priceProduct;  
    }  
  
    private double getPrice() {  
        return price;  
    }  
}
```



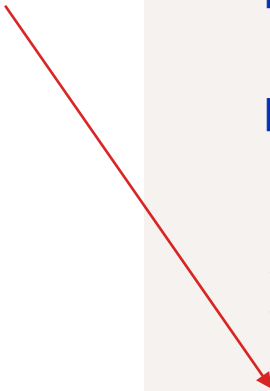
Решение

```
public class Practice {
    public static void main(String[] args) {
        Product product = new Product("Пирожок", 180.0);
        System.out.println("Стоимость продукта - " +
            product.getPrice() + " тг.");
    }
}

public class Product {
    private String name;
    private double price;

    public Product(String nameProduct, double priceProduct) {
        name = nameProduct;
        price = priceProduct;
    }

    public double getPrice() {
        return price;
    }
}
```





default или package-private


Модификатор доступа может быть не указан. В этом случае срабатывает модификатор по умолчанию — его еще называют **default** (англ. — «по умолчанию») или **package-private** (англ. — «приватный пакет»). Из второго названия понятно, что он даёт доступ к данным внутри своего пакета — за его пределами класс или метод уже не будут видны.

Если убрать у метода `calculate()` модификатор доступа и попробовать вызвать его сначала в его же пакете `calculator` — ошибки не будет.

```
package com.practice.calculator; // пакет calculator

public class WorkCalculator {
    private int workingHours = 8;

    int calculate(int workDays) { // модификатор по умолчанию
        return workDays * workingHours;
    }
}
```



```
package com.practice.calculator; // здесь тоже пакет calculator

public class Main {
    public static void main(String[] args) {
        WorkCalculator calculator = new WorkCalculator();
        System.out.println("Количество отработанных часов за 29 дней: "
            + calculator.calculate(29)); // метод сработал
    }
}
```



Результат

Количество отработанных часов за 29 дней: 232

При вызове из пакета `calendar` метод `calculate()` будет уже недоступен:

```
package com.practice.calendar; // здесь другой пакет
import com.practice.calculator.WorkCalculator;

public class Main {
    public static void main(String[] args) {
        WorkCalculator calculator = new WorkCalculator();
        System.out.println("Количество отработанных часов за 29 дней: "
            + calculator.calculate(29)); // метод сработал
    }
}
```



Результат

```
Main.java:3:29  
java: com.practice.calculator.WorkCalculator is not public in  
com.practice.calculator; cannot be accessed from outside package
```



protected

Ещё один, четвёртый, модификатор доступа **protected** (англ. «защищенный») схож по своему действию с модификатором по умолчанию. Поля или методы, помеченные им, также видны всем классам внутри пакета. Но помимо этого у **protected** есть дополнительное свойство — он даёт доступ к данным всем классам-наследникам, в том числе в разных пакетах. О наследовании и классах-наследниках мы подробно поговорим дальше.

Наглядно области видимости всех модификаторов доступа демонстрирует такая таблица.

Область видимости модификаторов доступа

Модификатор доступа	В классе	В пакете	В классе-наследнике	Везде	
public	●	●	●	●	✓
protected	●	●	●	●	✗
package-private (default)	●	●	●	●	
private	●	●	●	●	



Модификаторы доступа

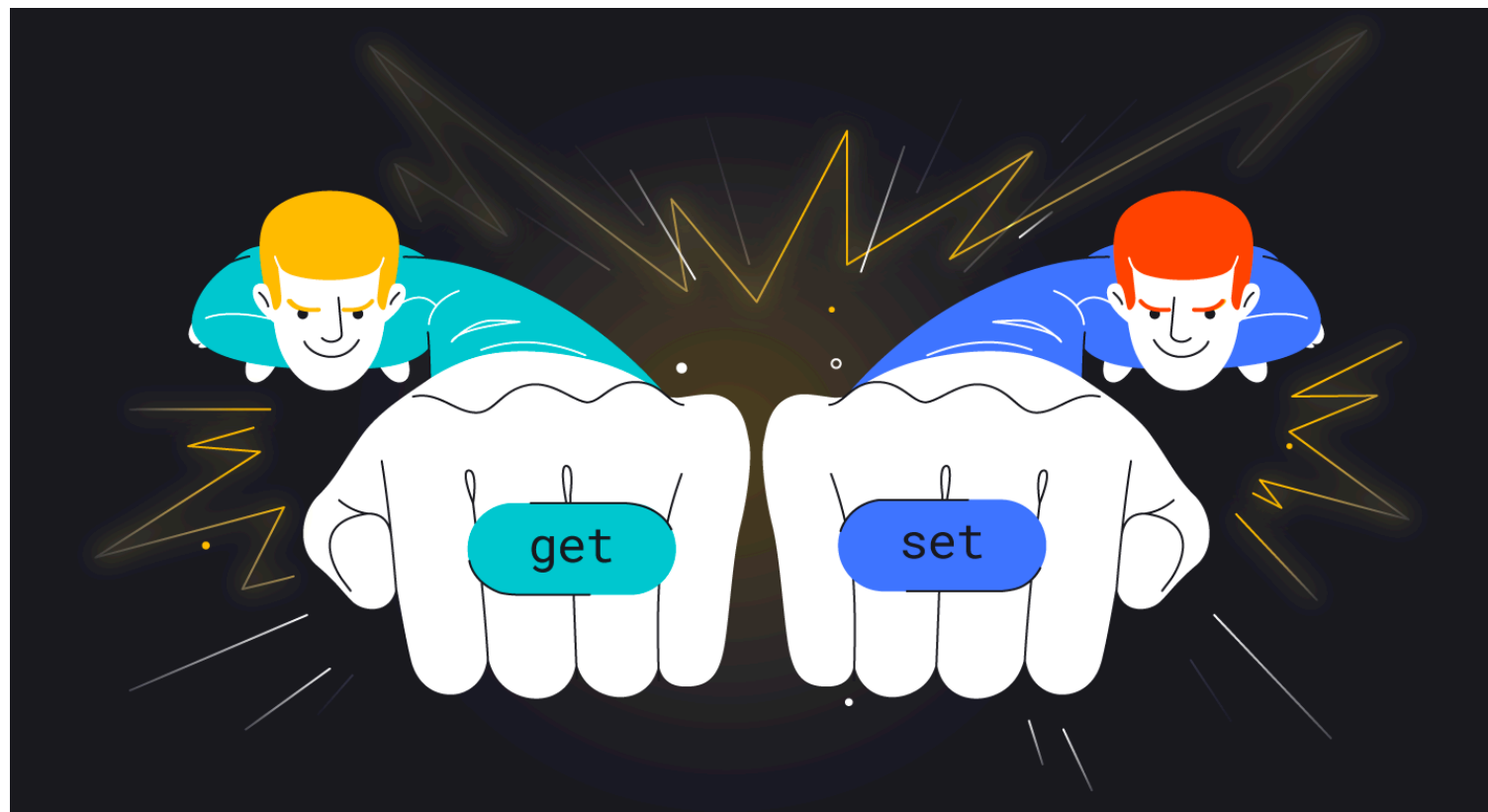
Забавное совпадение — названия всех модификаторов начинаются с одной буквы — английской «P». Вы можете легко придумать множество мнемонических формул, чтобы их запомнить. Например: «*Приватный (**private**) пакет (**package-private**) под защитой (**protected**) от публики (**public**)*».



Геттеры и сеттеры

Геттеры и сеттеры

Ещё один инструмент, который позволяет реализовать принцип инкапсуляции наравне с модификаторами доступа и пакетами — **get-** и **set-методы**. Их называют геттеры и сеттеры (англ. “getters and setters”).



Геттеры и сеттеры нужны для работы с полями класса, закрытыми модификатором **private**. К примеру, вам нужно присвоить новое значение полю **money** (англ. «деньги») в классе **Bank**. При этом доступ к переменной ограничен:

```
public class Bank {  
    private long money; // закрыли поле модификатором private  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Bank bank = new Bank();  
        bank.money = 2_342_345_221_223L; // ошибка - нет доступа к переменной  
    }  
}
```


Переменная **money** под надёжной защитой модификатора **private**. Если открыть доступ к ней — поставить модификатор **public**, то в **money** можно будет сохранить любое значение, в том числе отрицательное:

```
public class Bank {  
    public long money;  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Bank bank = new Bank();  
        bank.money = -9999; // Деньги украдены  
    }  
}
```



Геттеры и сеттеры

Чтобы взаимодействовать с защищённой переменной, но при этом не открывать к ней доступ, и используются **get**- и **set**-методы. **Get**-методы (от англ. *get* — получать) позволяют получать значения из закрытых переменных, а **set**-методы (от англ *set* — установка) сохранять в такие переменные новые значения.


В названии таких методов принято указывать слова **get** или **set** и имя переменной. Объявим их для переменной **money**:

```
public class Bank {  
    private long money = 0;  
  
    // создаём get-метод - с помощью него сможем получить значение money  
    public long getMoney() {  
        return money;  
    }  
  
    // создаём set-метод - он позволяет сохранить в money новое значение  
    public void setMoney(long newMoneyAmount) {  
        if (newMoneyAmount > 0) { // можно сохранить только положительные значения  
            money = newMoneyAmount;  
        }  
    }  
}
```



Геттеры и сеттеры

Код внутри **get**- и **set**-методов может быть любым. С его помощью может быть реализована дополнительная логика: обновление других переменных, вывод в консоль информации и многое другое. На внешнем интерфейсе это не отразится. То, насколько сложные операции происходят внутри методов, не должно затрагивать интересы пользователей.



Усложним код класса **Bank**. Добавим в него поле **commission** (англ. «комиссия»). Комиссию будем вычитать из той суммы, которую пользователь хочет положить на счёт. Для сумм до 25000 тенге предусмотрим сокращение комиссии в два раза:

```

public class Bank {
    private long money = 0;
    private int commission = 500; // доступ к полю ограничен

    public long getMoney() {
        return money;
    }

    public void setMoney(long newMoneyAmount) {
        calculateCommission(newMoneyAmount); // усложняем логику
        if (newMoneyAmount > 0 && newMoneyAmount > commission) {
            money = newMoneyAmount - commission;
        } else {
            commission = 0;
            System.out.println("Минимальная сумма - 251 тенге.");
        }
    }


    // Добавляем get-метод - комиссию можно будет показать пользователю
    public int getCommission() {
        return commission;
    }

    private void calculateCommission(long newMoneyAmount) {
        if (newMoneyAmount < 25000) {
            commission = 250;
        } else {
            commission = 500;
        }
    }
}

```

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Bank bank = new Bank();
        Scanner scan = new Scanner(System.in);
        System.out.println("Сколько вы хотите положить на счёт?");
        long money = scan.nextLong();
        bank.setMoney(money);
        System.out.println("На вашем счету " + bank.getMoney() + " тенге.");
        System.out.println("Комиссия составила " + bank.getCommission() + " тенге.");
    }
}
```



С помощью метода **getCommission()** пользователь может получить значение комиссии. Для её расчёта в программу добавлен метод **calculateCommission()**. Логика установки суммы на счету в методе **setMoney()** при этом усложнилась.

Для получения значений из переменных типа **boolean** вместо **get** используется глагол **is** (для установки значений также используется префикс **set**-). Например, добавим в код логическую переменную **isOfficial**, которая отражает, является ли банк государственным:


```
public class Bank {  
    private long money = 0;  
    private int commission = 500;  
    private boolean isOfficial = false; // является ли банк государственным  
  
    public boolean isOfficial() { // метод называется так же, как переменная  
        return isOfficial;  
    }  
  
    public void setOfficial(boolean newOfficial) {  
        isOfficial = newOfficial;  
    }  
  
    public long getMoney() {  
        return money;  
    }  
  
    public void setMoney(long newMoneyAmount) {  
        if (newMoneyAmount > 0) {  
            money = newMoneyAmount;  
        }  
    }  
}
```



Геттеры и сеттеры

Геттеры и сеттеры ничем и не отличаются от обычных методов. Можно прописать получение и установку значений закрытых переменных с помощью методов с любыми другими именами, например, `addMoney()`, `receiveMoney()`, `takeMoney()` или `winMoney()`, и программа скомпилируется. Однако любая нотация кроме `get-` и `set-` считается некорректной, так как не позволяет разработчику быстро понять, с чем он работает, а заставляет дополнительно разбираться в логике работы методов.



Какие из этих названия методов **get-** и **set-** правильны для переменной `private int voiceVolume`

- A. `getVoiceVolume()`
- B. `setvoiceVolume()`
- C. `isVoiceVolume()`
- D. `setVoiceVolume()`
- E. `addVoiceVolume()`

Ответ

A. `getVoiceVolume()`

Один из вариантов — использовать префикс **get** и **Camel**-нотацию.

B. `setvoiceVolume()`

Префикс **set**- — верный, но после него, согласно **Camel**-нотации, должна идти заглавная буква.

C. `isVoiceVolume()`

Префикс **is** нужен для булевых переменных.

D. `setVoiceVolume()`

Префикс **set** и **Camel**-нотация дают нам корректный метод для установки значений типа **int**.

E. `addVoiceVolume()`

Для установки нового значения переменной требуется **set**-метод.

Задача

Вам нужно снять наличные в банкомате, но он сломался и выводит только консоль с недописанным кодом. По счастливой случайности — на Java. Допишите код — реализуйте методы в классе **BankAccount**. Чтобы установить и считать значение суммы денег на счёте **moneyAmount**, вам понадобятся get- и set-методы. Чтобы снять деньги со счёта и обнулить его — метод **withdrawAll()**, который должен обнулять счёт и печатать количество выданных денег в формате: *Со счёта снято 10000 тг.* Все методы должны иметь самый широкий уровень доступа. В результате запуска программы в консоли должно появиться:

https://github.com/practicetasks/java_tasks/tree/main/encapsulation/task_1



Решение

<https://gist.github.com/practicetasks/3c0c5149823b24b7daee79e304b8bfab>



Наследование



Наследование

Перейдём от инкапсуляции к другому принципу ООП — **наследованию** (от англ. “inheritance”). Ранее мы упоминали об одном из главных правил разработки DRY — Don’t repeat yourself (англ. — «не повторяйся»). Принцип наследования как раз призван решить проблему повторения кода.



Наследование

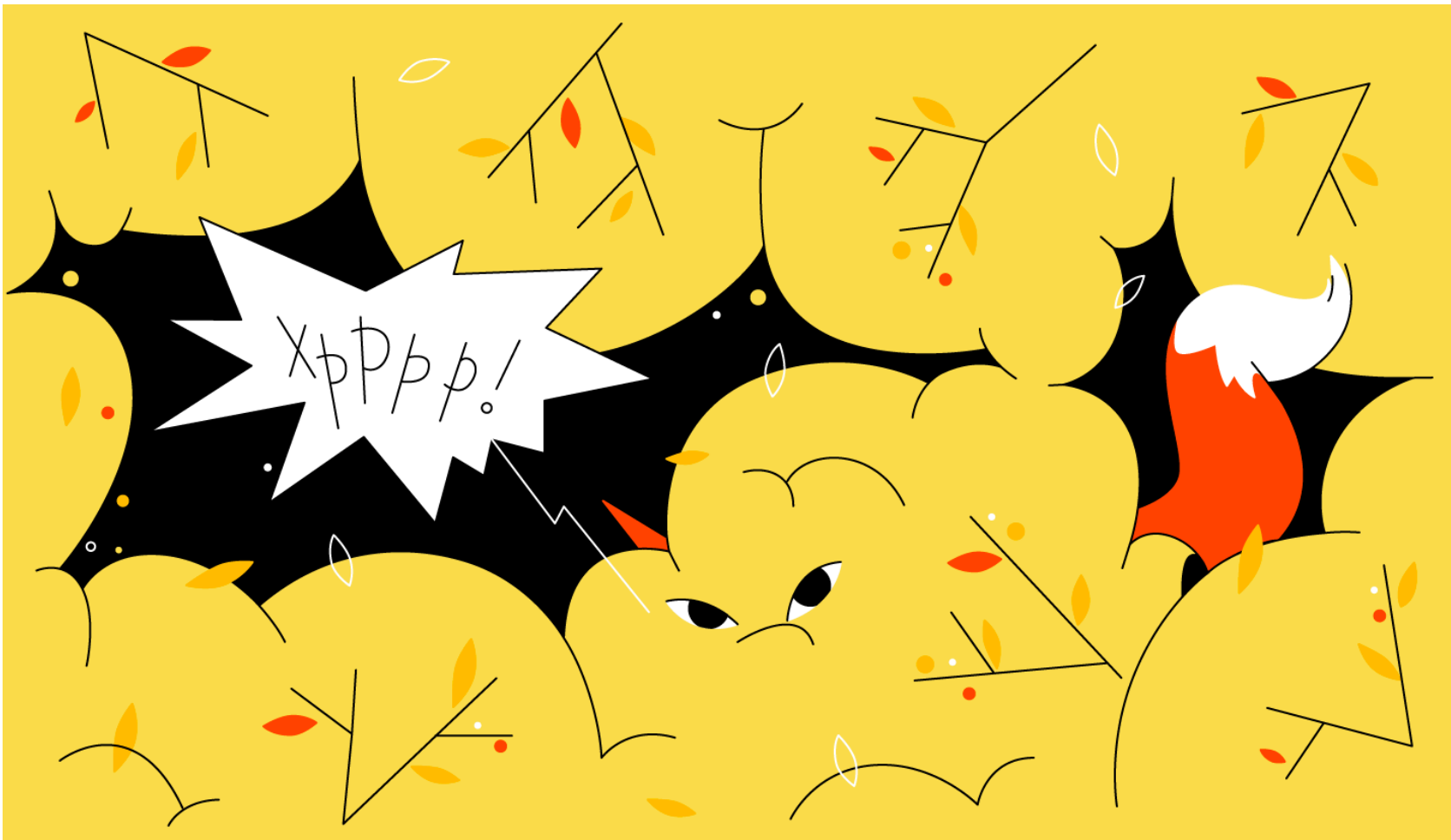
Благодаря наследованию дочерние классы автоматически приобретают функционал класса-родителя. Не нужно раз за разом прописывать одни и те же поля и методы — их можно передать по наследству. Разберём, как это работает.

Возьмём простой пример — на уроках биологии в школе вы изучали классификацию видов в природе. Она построена по иерархическому принципу: живые организмы делятся по типам, типы по классам, классы по отрядам и так далее.

Чтобы отобразить многообразие животного мира в программе, следует начать с общего класса **Animal** (англ. «животное»):

```
public class Animal {  
    protected double weight; // вес животного  
    protected int age; // возраст животного  
    protected int heartRate; // количество ударов сердца в минуту  
    protected boolean isWild; // является ли оно диким  
  
    public Animal() {  
        weight = 0.0;  
        age = 0;  
        heartRate = 100;  
        isWild = true;  
    }  
  
    public String say() {  
        return "Я животное!";  
    }  
}
```

В получившемся классе **Animal** определено некоторое количество полей, идентичных для разных животных. У всех животных есть вес и возраст, бьётся сердце, их можно разделить на одомашненных и тех, кто живёт в дикой природе. При этом пока объект класса **Animal** может сообщить, что он животное, только при помощи метода **say()**.




Чтобы передать все характеристики класса **Animal** другому классу, например, **Canidae**(от англ. «псовые»), нужно применить принцип наследования. Для этого требуется ключевое слово **extends**:

```
public class Canidae extends Animal { // здесь происходит наследование
    protected boolean isPredator; // добавляем новое поле: хищник или нет

    public Canidae() { // добавляем конструктор
        isPredator = true;
    }

    public String growl() { // добавляем новый метод - рычание
        return "P-p-p-p-p!";
    }
}
```



Класс **Canidae** наследует все поля и методы класса **Animal**, при этом у него появляются свои, которые также можно передать по наследству:

```
public class Fox extends Canidae { // здесь происходит наследование и Canidae, и Animal
    protected String color; // добавляем новое поле
}
```

Важно запомнить, что у классов в Java может быть сколько угодно предков, но только один родитель. То есть при помощи ключевого слова **extends** можно наследовать только от одного класса. Написать `class Fox extends Animal, Canidae` не получится.



Вернёмся к примеру. Созданный нами класс **Fox** наследник и **Canidae**, и **Animal**. Его объекты могут иметь рост, вес и кричать «Я животное!», быть хищниками и рычать, а также отличаться своим цветом. При этом нам не пришлось трижды дублировать один и тот же код. Посмотрим, как это работает — создадим объект класса **Fox**:

```
public class Animal {  
    protected double weight;  
    protected int age;  
    protected int heartRate;  
    protected boolean isWild;  
  
    public Animal() {  
        weight = 0.0;  
        age = 0;  
        heartRate = 100;  
        isWild = true;  
    }  
  
    public String say() {  
        return "Я животное!";  
    }  
}
```

```
public class Canidae extends Animal { // здесь происходит наследование
    protected boolean isPredator;

    public Canidae() {
        isPredator = true;
    }

    public String growl() {
        return "P-p-p-p-p!";
    }
}

public class Fox extends Canidae { // здесь происходит наследование и Canidae, и Animal
    protected String color;

    public Fox() {
        color = "рыжий";
        weight = 2.0;
    }
}
```



```
public class Practice {  
    public static void main(String[] args) {  
        Fox foxAlica = new Fox();  
        System.out.println("Это лиса Алиса");  
        System.out.println("Цвет - " + foxAlica.color); // поле класса Fox  
        System.out.println("Вес - " + foxAlica.weight + " кг"); // конструктор Fox  
        System.out.println("Она дикая - " + foxAlica.isWild); // поле класса Animal  
        System.out.println("Хищник - " + foxAlica.isPredator); // поле класса Canidae  
        System.out.println("Она умеет говорить - " + foxAlica.say()); // метод класса Animal  
        System.out.println("И может зарычать " + foxAlica.growl()); // метод класса Canidae  
    }  
}
```



Результат

Это лиса Алиса
Цвет - рыжий
Вес - 2.0 кг
Она дикая - true
Хищник - true
Она умеет говорить - Я животное!
И может зарычать Р-р-р-р-р!

Корректные определения наследования в Java

- A. Это возможность языка усложняться и развиваться, чтобы на нём было удобнее писать код.
- B. Это возможность передавать функционал и характеристики одного класса другому с помощью ключевого слова **extends**.
- C. Это возможность дать одним классам доступ к функционалу и характеристикам других классов.
- D. Это возможность расширить функционал новых классов за счёт уже написанных без необходимости дублировать код.
- E. Это возможность объединить и структурировать похожие классы.

А. Это возможность языка усложняться и развиваться, чтобы на нём было удобнее писать код.

Развитие языка и удобство написания кода на нём не определяет, что такое наследование.

В. Это возможность передавать функционал и характеристики одного класса другому с помощью ключевого слова **extends**.

Именно ключевое слово **extends** отвечает за наследование в Java.

С. Это возможность дать одним классам доступ к функционалу и характеристикам других классов.

Это определение инкапсуляции — она отвечает за то, чтобы открыть или закрыть к чему-то доступ.

Д. Это возможность расширить функционал новых классов за счёт уже написанных без необходимости дублировать код.

Именно это мы и сделали с классом Fox.

Е. Это возможность объединить и структурировать похожие классы.

Для структурирования классов в Java используются другие инструменты — например пакеты.



Вступаем в наследство

Класс-родитель, или класс-предок, иначе ещё называют **суперклассом**. Классы-потомки — **подклассами**. Разберём подробнее, что и как можно передавать по наследству из суперкласса в подкласс, и может ли родитель получать что-то от своих наследников.

Подкласс наследует все поля и методы суперкласса, которые имеют области видимости **public**, **protected** или **package-private** (если класс-потомок находится в одном пакете с классом-родителем). Данные, помеченные модификатором **private**, при расширении с помощью **extends** не передаются.





Что можно делать в подклассе

После расширения с помощью **extends** в классе-наследнике можно делать следующее:

- Использовать унаследованные поля и методы напрямую.

```
public class Coffee {
    protected double espresso;
    protected double milk;

    public Coffee() {
        espresso = 50.0;
        milk = 250.0;
    }

    public double mixAndGetVolume() {
        return espresso + milk;
    }
}

public class Cappuccino extends Coffee {
}

public class Latte extends Coffee {
}

public class Practice {
    public static void main(String[] args) {
        Cappuccino cappuccino = new Cappuccino();
        // напрямую используем метод суперкласса:
        System.out.println("Объём чашки капучино - " + cappuccino.mixAndGetVolume());
        Latte latte = new Latte();
        // напрямую используем переменную суперкласса:
        System.out.println("Объём эспрессо в латте - " + latte.espresso);
    }
}
```




Что можно делать в подклассе

- Можно объявить новые поля и методы, которых нет в суперклассе.

```
public class Coffee {
    protected double espresso;
    protected double milk;

    public Coffee() {
        espresso = 50.0;
        milk = 250.0;
    }

    public double mixAndGetVolume() {
        return espresso + milk;
    }
}

public class Cappuccino extends Coffee {
    private double milkFoam; // добавили поле

    public double getMilkFoam() { // добавили метод
        milkFoam = milk / 2; // использовали переменную суперкласса
        return milkFoam;
    }
}

public class Practice {
    public static void main(String[] args) {
        Cappuccino cappuccino = new Cappuccino();
        System.out.println("Объём молочной пены - " + cappuccino.getMilkFoam());
    }
}
```



Что можно делать в подклассе

- Можно объявить поле в подклассе с таким же именем, что и поле в суперклассе, — это называется **сокрытием** (англ. *hiding*). Увлекаться им не стоит — в результате получается два класса, один из которых наследует от другого и в обоих есть одинаковые поля. Если произойдёт ошибка — найти её будет сложно.

```
public class Coffee {
    protected double espresso;
    protected double milk;

    public Coffee() {
        espresso = 50.0;
        milk = 250.0;
    }

    public double mixAndGetVolume() {
        return espresso + milk;
    }
}

public class Cappuccino extends Coffee {
    protected double espresso = 30.0;
}

public class Practice {
    public static void main(String[] args) {
        Cappuccino cappuccino = new Cappuccino();
        System.out.println("Объём эспрессо - " + cappuccino.espresso);
    }
}
```



Что можно делать в подклассе

- Можно **переопределить методы**: реализовать в подклассе методы, которые будут иметь ту же сигнатуру, что и в суперклассе, но отличаться своим поведением.

```
public class Coffee {
    protected double espresso;
    protected double milk;

    public Coffee() {
        espresso = 50.0;
        milk = 250.0;
    }

    public double mixAndGetVolume() {
        return espresso + milk;
    }
}

public class Cappuccino extends Coffee {
    public double mixAndGetVolume() { // переопределили метод mixAndGetVolume()
        double milkFoam = milk / 2;
        return espresso + milkFoam + milk / 3;
    }
}

public class Practice {
    public static void main(String[] args) {
        Cappuccino cappuccino = new Cappuccino();
        System.out.println("Объём капучино - " + cappuccino.mixAndGetVolume());
    }
}
```



Что можно делать в подклассе

- Можно написать конструктор, который будет вызывать конструктор суперкласса.

```

public class Coffee {
    protected double espresso;
    protected double milk;

    public Coffee(double newEspresso, double newMilk) { // конструктор суперкласса
        espresso = newEspresso;
        milk = newMilk;
    }

    public double mixAndGetVolume() {
        return espresso + milk;
    }
}

public class Cappuccino extends Coffee {
    public Cappuccino() {
        //вызываем конструктор суперкласса
        super(100.0, 250.0); // о ключевом слове super поговорим подробно чуть позже
    }
}

public class Practice {
    public static void main(String[] args) {
        Cappuccino cappuccino = new Cappuccino();
        System.out.println("Объём капучино - " + cappuccino.mixAndGetVolume());
    }
}

```


Класс **Superhuman** наследует от **Human**, что можно сделать в таком случае внутри **Superhuman**?

A. Можно переопределить доступные методы класса **Human**.

B. Можно использовать все поля и методы класса **Human**, в том числе с модификатором **private**.

C. Можно добавить второго предка — класс **Power**.

D. Можно объявить в **Superhuman** новое поле **superstrength** и метод **lasers()**, которых нет в **Human**.

E. Можно написать конструктор **Superhuman**, который будет вызывать публичный конструктор **Human**.

F. Можно заблокировать использование полей и методов **Human** в других подклассах.

A. Можно переопределить доступные методы класса **Human**.

Если доступ к методам не закрыт модификатором **private** — их можно переопределить.

B. Можно использовать все поля и методы класса **Human**, в том числе с модификатором **private**.

Если у суперкласса есть скрытые **private**-модификатором поля или методы, то в подклассе их использовать не получится.

C. Можно добавить второго предка — класс **Power**.

В Java нельзя наследовать сразу от двух классов. Придётся выбирать — либо от **Human**, либо от **Power**.

D. Можно объявить в **Superhuman** новое поле **superstrength** и метод **lasers()**, которых нет в **Human**.

В подклассах можно легко создавать свои поля и методы.

E. Можно написать конструктор **Superhuman**, который будет вызывать публичный конструктор **Human**.

Если в родительском классе есть публичные конструкторы — мы можем их вызвать в классе-наследнике.

F. Можно заблокировать использование полей и методов **Human** в других подклассах.

Так сделать, к сожалению, не получится.

Связь с родителем

В наследовании есть одно важное правило — в экземплярах суперкласса нельзя использовать поля и методы подклассов. То есть если у классов-наследников появились новые свойства и функционал, то передать их в класс-родитель не получится.

Это правило отлично иллюстрируется эволюцией техники: на кнопочный телефон не получится установить ни одно из современных приложений.





Связь с родителем

Также в классе-наследнике нельзя сузить видимость полей или методов — можно только расширить. Например, если в суперклассе метод был **public**, то в подклассе он не может стать **private** или **protected**. Если же у родительского метода модификатор **protected**, то в наследнике его можно изменить на **public**.



Переопределяем методы

Переопределяем методы

Чтобы изменить поведение метода суперкласса, его можно **переопределить** внутри подклассов. Механизм переопределения предполагает, что сигнатура остаётся прежней, при этом в тело метода вносятся изменения, а доступ к нему может быть расширен.

Переопределение метода помечается в коде с помощью аннотации **@Override** (от англ. *override* — «переопределение, ручная коррекция»).

Аннотации в Java — это специальная форма метаданных в коде. Они начинаются с символа @ и сообщают компилятору дополнительную информацию. Например, если **@Override** помечает метод как переопределённый, то **@Deprecated** как устаревший. Аннотациями можно помечать методы, классы и переменные.

К примеру, для класса-родителя **Teacher** (англ. «учитель») и подкласса **GeographyTeacher** (англ. «учитель географии») актуален метод **startLesson()** (англ. «начать урок»). При этом если в суперклассе у него максимально общее содержание — «Достаём учебники!», то в класс-наследнике оно может быть конкретным — «Достаём глобусы!». Переопределим метод **startLesson()** внутри класса **GeographyTeacher**:

```
public class Teacher {  
    protected String startLesson() { // доступ ограничен классами-наследниками  
        return "Достаём учебники!";  
    }  
}  
  
public class GeographyTeacher extends Teacher {  
    @Override // аннотация  
    public String startLesson() { // доступ стал публичным  
        return "Достаём глобусы!"; // изменилось содержание  
    }  
}
```

Корректное описание переопределения

- A. Это особенность языка, позволяющая менять методы.
- B. Это особенность языка Java, связанная с наследованием методов.
- C. Это аннотация `@Override`.
- D. Это один из инструментов языка, позволяющий классу-наследнику обеспечивать специфическую реализацию метода класса-родителя.
- E. Это возможность объекта класса-наследника вызывать методы класса-родителя.

А. Это особенность языка, позволяющая менять методы.

Переопределение не про изменение всех методов, а про изменение поведения методов суперкласса в подклассе.

В. Это особенность языка Java, связанная с наследованием методов.

Переопределение встречается и в других языках программирования.

С. Это аннотация `@Override`.

Аннотация лишь сообщает, что мы переопределяем метод суперкласса.

Д. Это один из инструментов языка, позволяющий классу-наследнику обеспечивать специфическую реализацию метода класса-родителя.

Именно для этого и нужно переопределение в Java.

Е. Это возможность объекта класса-наследника вызывать методы класса-родителя.

При вызове переопределённого метода будет задействована реализация из класса-наследника:

```
public class Teacher {  
    public String startLesson() {  
        return "Достаём учебники!";  
    }  
}  
  
public class GeographyTeacher extends Teacher {  
    @Override  
    public String startLesson() {  
        return "Достаём глобусы!";  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        // создали объект подкласса  
        GeographyTeacher teacher = new GeographyTeacher();  
        // вызываем метод подкласса  
        System.out.println("Урок начнётся с фразы : " + teacher.startLesson());  
    }  
}
```



Результат

Урок начнётся с фразы : Достаем глобусы!

Исправьте метод `playHamlet()` так, чтобы исполнитель `performer` прочитал строчку из монолога Гамлета — «Быть или не быть? Вот в чём вопрос?».

```
public class Actor {
    void play() {
        System.out.println("Гул затих. Я вышел на подмостки.");
    }
}

public class Hamlet extends Actor {
    void playHamlet() {
        System.out.println("Быть или не быть? Вот в чём вопрос?");
    }
}

public class Practice {
    public static void main(String[] args) {
        Hamlet performer = new Hamlet();
        performer.play();
    }
}
```

Решение

```
public class Actor {
    void play() {
        System.out.println("Гул затих. Я вышел на подмостки.");
    }
}

public class Hamlet extends Actor {
    @Override
    void play() {
        System.out.println("Быть или не быть? Вот в чём вопрос?");
    }
}

public class Practice {
    public static void main(String[] args) {
        Hamlet performer = new Hamlet();
        performer.play();
    }
}
```



Плюсы аннотации `@Override`

Отсутствие аннотации `@Override` при переопределении метода — не ошибка. Однако её принято использовать, так как у неё есть два полезных свойства:

- Явно обозначены переопределённые методы — их легко отличить от остальных методов класса.
- Если в переопределённом методе, помеченном `@Override`, поменяется сигнатура (неважно где: в классе-родителе или классе-наследнике), то при компиляции появится сообщение об этом. *Method does not override from its superclass* — это означает, что метод больше не переопределяется из своего суперкласса.

```
public class Teacher {
    public String getHomework(String lessonName) {
        return "Читать пятый параграф!";
    }
}

public class GeographyTeacher extends Teacher {
    // ошибка компиляции
    @Override
    public String getHomework() {
        return "Выучить столицы всех стран.";
    }
}

public class Practice {
    public static void main(String[] args) {
        GeographyTeacher teacher = new GeographyTeacher();
        System.out.println("Задание на дом: " + teacher.getHomework()); // тоже ошибка
    }
}
```



Плюсы аннотации `@Override`

Важно помнить, что наследование крепко связывает классы: если понадобится добавить в метод суперкласса новый параметр, а от него уже созданы десятки классов-наследников — придётся вносить изменения во все переопределённые методы! Аннотация `@Override` здесь станет отличным помощником.

Зачем нужна аннотация `@Override`.

- A. Подсказывает компилятору, что мы пытаемся переопределить метод класса-родителя.
- B. При изменении типов или количества параметров метода суперкласса поможет найти ошибку в компиляции.
- C. Обязательна при переопределении методов.
- D. Вызывает родительский метод перед исполнением метода класса-наследника.
- E. Упрощает чтение кода: показывает, что метод переопределяет метод класса-родителя.

А. Подсказывает компилятору, что мы пытаемся переопределить метод класса-родителя. Если метод не будет найден в суперклассе — появится сообщение об ошибке.

В. При изменении типов или количества параметров метода суперкласса поможет найти ошибку в компиляции.

Это одна из основных причин, по которой рекомендуется всегда использовать `@Override` с переопределёнными методами.

С. Обязательна при переопределении методов.

Можно и не использовать аннотацию, но так делать не рекомендуется.

Д. Вызывает родительский метод перед исполнением метода класса-наследника.

Такой функционал противоречил бы переопределению.

Е. Упрощает чтение кода: показывает, что метод переопределяет метод класса-родителя.

Так можно увидеть, что такой же метод есть в суперклассе.



Комбинация **Ctrl+O**

При работе в среде разработки IDEA, чтобы переопределить один или несколько методов в подклассе, можно использовать комбинацию **Ctrl+O** (для Windows и Linux) или **^ (Control) + O** (для Mac OS X). При её использовании легко сразу выбрать все методы, поведение которых нужно изменить. Также IDEA заботливо добавит аннотацию **@Override** к этим методам.



Ключевое слово — super



Ключевое слово — `super`

Было бы нелогичным, если бы после переопределения методов в подклассе мы лишались возможности обратиться к первоисточнику. Для этого существует ключевое слово **`super`** — с его помощью можно вызвать метод или конструктор суперкласса, а также обратиться к его полям.



Суперметоды

Для обращения к методам класса-родителя через **super** нужно применить точечную нотацию — **super.someMethod()**. Это может быть полезно, когда нужно использовать функционал родительского метода и дополнить его новыми действиями.



Суперметоды

К примеру, учитель физкультуры (**GymTeacher**) может начинать урок с общей для всех учителей (**Teacher**) фразы «Звонок для кого прозвенел?!». И только после этого переходить к более конкретному указанию: «Стройтесь по росту!». Чтобы реализовать это в коде, нужно обратиться к родительскому методу из переопределяемого метода класса-наследника:

```
public class Teacher {  
    public void startLesson() {  
        System.out.println("Звонок для кого прозвенел?!");  
    }  
}  
  
public class GymTeacher extends Teacher {  
    @Override  
    public void startLesson() {  
        super.startLesson(); //вызов метода класса-родителя  
        System.out.println("Стройтесь по росту!");  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        GymTeacher teacher = new GymTeacher();  
        System.out.println("Учитель физкультуры говорит:");  
        teacher.startLesson();  
    }  
}
```




Результат

Учитель физкультуры говорит:
Звонок для кого прозвенел?!
Стройтесь по росту!



Суперполя

С помощью ключевого слова **super** и точечной нотации можно обратиться также к скрытым полям родительского класса — **super.someField**.



Суперполя

К примеру, поле `numberOfLessons` класса-родителя `Teacher`, где задано общее количество уроков на неделе, было скрыто в подклассе `GymTeacher` — в него были сохранены уроки физкультуры. Напечатаем значения обоих полей:

```
public class Teacher {
    int numberOfLessons = 34; // поле класса-родителя скрыто
}

public class GymTeacher extends Teacher {
    int numberOfLessons = 3;

    public void printSchedule() {
        // печатаем значение поля суперкласса
        System.out.println("Число уроков в неделю - " + super.numberOfLessons);
        // печатаем значение поля подкласса
        System.out.println("Число уроков физкультуры - " + numberOfLessons);
    }
}

public class Practice {
    public static void main(String[] args) {
        GymTeacher teacher = new GymTeacher();
        teacher.printSchedule();
    }
}
```



Результат

Число уроков в неделю - 34

Число уроков физкультуры - 3

Необходимость обращаться через **super** к скрытым полям суперкласса — достаточно редкое явление. Во-первых, потому что увлекаться сокращением полей не принято — так возрастает вероятность ошибки. Во-вторых, поля классов всё-таки лучше инициализировать не при объявлении, а в конструкторе.



Суперконструкторы

Через **super** можно также вызвать конструктор класса-родителя. С помощью **super()** — без параметров, а с помощью **super(parameter list)** — с параметрами. **parameter list** подразумевает, что нужно передать определённое число параметров указанных типов.



super()

Если в классе-родителе есть конструктор без параметров, то в конструкторе подкласса компилятор вызовет его автоматически, неявно.

К примеру, число учеников в классе (**numberOfPupils**) и норма нагрузки (**workLoad**) для всех учителей одинаковая — значит, значения этим полям можно присвоить в конструкторе класса-родителя **Teacher**. Тогда в конструкторе подкласса **BiologyTeacher** (англ. «учитель биологии») нужно будет только проинициализировать индивидуальные поля, например, количество лабораторных работ (**numberOfLabs**):

```

public class Teacher {
    int numberOfPupils; // число учеников в классе
    double workload; // нагрузка учителя

    public Teacher() {
        numberOfPupils = 30;
        workload = 22;
    }
}

public class BiologyTeacher extends Teacher {
    int numberOfLabs; // число лабораторных работ

    public BiologyTeacher() {
        // здесь неявно вызван конструктор суперкласса
        numberOfLabs = 10;
    }
}

public class Practice {
    public static void main(String[] args) {
        BiologyTeacher teacher = new BiologyTeacher();
        System.out.println("Число учеников на уроке биологии - " + teacher.numberOfPupils);
        System.out.println("Рабочая нагрузка " + teacher.workload + " часов в месяц");
        System.out.println("Количество лабораторных работ - " + teacher.numberOfLabs);
    }
}

```




Результат

Число учеников на уроке биологии - 30
Рабочая нагрузка 22.0 часов в месяц
Количество лабораторных работ - 10



`super()`

Если явно вызвать конструктор суперкласса, то код получится таким:

```

public class Teacher {
    int numberOfPupils; // число учеников в классе
    double workload; // нагрузка учителя

    public Teacher() {
        numberOfPupils = 30;
        workload = 22;
    }
}

public class BiologyTeacher extends Teacher {
    int numberOfLabs; // число лабораторных работ

    public BiologyTeacher() {
        super(); // здесь явно вызван конструктор суперкласса
        numberOfLabs = 10;
    }
}

public class Practice {
    public static void main(String[] args) {
        BiologyTeacher teacher = new BiologyTeacher();
        System.out.println("Число учеников на уроке биологии - " + teacher.numberOfPupils);
        System.out.println("Рабочая нагрузка " + teacher.workload + " часов в месяц");
        System.out.println("Количество лабораторных работ - " + teacher.numberOfLabs);
    }
}

```



`super()`

Явно вызывать конструктор суперкласса без параметров — нет необходимости. Единственное исключение: для навигации в коде — зажав **ctrl** и кликнув на **super()**, можно быстро перейти в конструктор родителя.



`super(parameter list)`

Совсем другая история — наличие в классе-родителе конструктора с параметрами. В этом случае обойтись без вызова суперконструктора в классе-наследнике не получится — произойдёт ошибка компиляции.

```
public class Teacher {  
    String name;  
    // конструктора без параметров (default) нет  
    public Teacher(String newName) { // объявлен конструктор с параметром  
        name = newName;  
    }  
}  
  
public class LiteratureTeacher extends Teacher {  
    public LiteratureTeacher() { // тут произойдёт ошибка компиляции  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        LiteratureTeacher teacher = new LiteratureTeacher();  
    }  
}
```

Результат

```
Teacher in class Teacher cannot be applied to given types;  
required: java.lang.String  
found:    no arguments  
reason: actual and formal argument lists differ in length
```

IDEA в таком случае подскажет, что **"There is no default constructor available in 'Teacher'"** — англ. «в **Teacher** нет доступного конструктора по умолчанию». А при попытке запустить код компилятор выдаст ошибку: **"constructor Teacher in class Teacher cannot be applied to given types"** — англ. «конструктор **Teacher** в классе **Teacher** не может быть применён к заданным типам».

Чтобы решить эту проблему, нужно либо создать в суперклассе конструктор без параметров, либо вызвать родительский конструктор:

```
public class Teacher {
    String name;

    public Teacher(String newName) {
        name = newName;
    }
}

public class LiteratureTeacher extends Teacher {
    public LiteratureTeacher() {
        super("Мария Ивановна Петрова"); // вызвали родительский конструктор
    }
}

public class Practice {
    public static void main(String[] args) {
        LiteratureTeacher teacher = new LiteratureTeacher();
        // объект создан в классе-наследнике при помощи конструктора класса-родителя
        System.out.println("Учитель литературы в 11-Б - " + teacher.name);
    }
}
```




Результат

Учитель литературы в 11-Б - Мария Ивановна Петрова



`super(parameter list)`

Чтобы не дублировать код конструктора родителя в классе-наследнике, можно вызвать суперконструктор в конструкторе подкласса. Тогда общие поля будут проинициализированы в классе-родителе, а индивидуальные — в наследнике. Например:

```
public class Teacher {  
    int numberOfLessons; // число уроков  
    int numberOfPupils; // число учеников  
    String name; // имя учителя  
  
    // конструктор класса-родителя  
    public Teacher(int newNumberOfLessons,  
                   int newNumberOfPupils,  
                   String newName) {  
        numberOfLessons = newNumberOfLessons;  
        numberOfPupils = newNumberOfPupils;  
        name = newName;  
    }  
}
```

```

public class GymTeacher extends Teacher {
    int basketballLessons; // уроки по баскетболу
    int swimmingLessons; // уроки по плаванию
    int freeLessons; // уроки без строгой программы

    // конструктор в подклассе принимает и свои параметры, и параметры суперкласса
    public GymTeacher(int newNumberOfLessons,
                      int newNumberOfPupils,
                      String newName,
                      int numberBasketballLessons,
                      int numberSwimmingLessons) {
        // сначала вызываем конструктор класса-родителя
        super(newNumberOfLessons, newNumberOfPupils, newName);
        // инициализируем новые поля
        basketballLessons = numberBasketballLessons;
        swimmingLessons = numberSwimmingLessons;
        // поле freeLessons вычисляется на основе заданных параметров
        freeLessons = newNumberOfLessons - numberBasketballLessons - numberSwimmingLessons;
    }

    public void printInfo() {
        System.out.println("ФИО учителя - " + name);
        System.out.println("Число учеников - " + numberOfPupils);
        System.out.println("Число уроков - " + numberOfLessons);
        System.out.println("Уроков по плаванию - " + swimmingLessons);
        System.out.println("Уроков по баскетболу - " + basketballLessons);
        System.out.println("Свободных уроков - " + freeLessons);
    }
}

```

```
public class Practice {  
    public static void main(String[] args) {  
        GymTeacher teacher = new GymTeacher(12, 30, "Олег Евгеньевич Фомин", 5, 4);  
        teacher.printInfo();  
    }  
}
```

Результат

ФИО учителя - Олег Евгеньевич Фомин
Число учеников - 30
Число уроков - 12
Уроков по плаванию - 4
Уроков по баскетболу - 5
Свободных уроков - 3

В конструкторе с параметрами класса-родителя **Teacher** уже определены три поля. Чтобы не дублировать их инициализацию в конструкторе наследника **GymTeacher**, достаточно вызвать конструктор через **super(parameters list)**. Затем можно добавить новые поля.



super()

Важное правило! Вызов конструктора класса-родителя через **super** должен быть первой строкой в конструкторе класса-наследника. Иначе произойдёт ошибка "**java: call to super must be first statement in constructor**" — англ. «вызов **super** должен быть первым оператором в конструкторе». Так компилятор проверяет, что родительский класс был проинициализирован корректно ещё до создания дочернего класса.



Добавляем `this` в конструкторы



Добавляем `this` в конструкторы

Материал уроков усложняется — в коде примеров и тренажёров появляется всё больше разных классов. Для инициализации полей и создания объектов не обойтись без конструкторов. Вы уже немного знакомы с этим инструментом.

Почему без конструктора плохо

Если не объявить в классе конструктор — Java сгенерирует конструктор по умолчанию. У него нет параметров, в коде его не видно, но именно он вызывается при создании объекта с помощью **new**.

```
public class Bot {  
    String name = "Элиза"; // проинициализировали поле при объявлении  
  
    // конструктор по умолчанию не видно, но он есть  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        Bot bot = new Bot(); // конструктор вызывается после new  
        System.out.println("Вас приветствует чат-бот " + bot.name);  
    }  
}
```



Результат

Вас приветствует чат-бот Элиза



Почему без конструктора плохо

Инициализировать поля при объявлении и обходиться только конструктором по умолчанию — можно, но не принято. Такой код неудобно читать — особенно если в классе много полей.

Можно создать конструктор без параметров и проинициализировать поля внутри него:

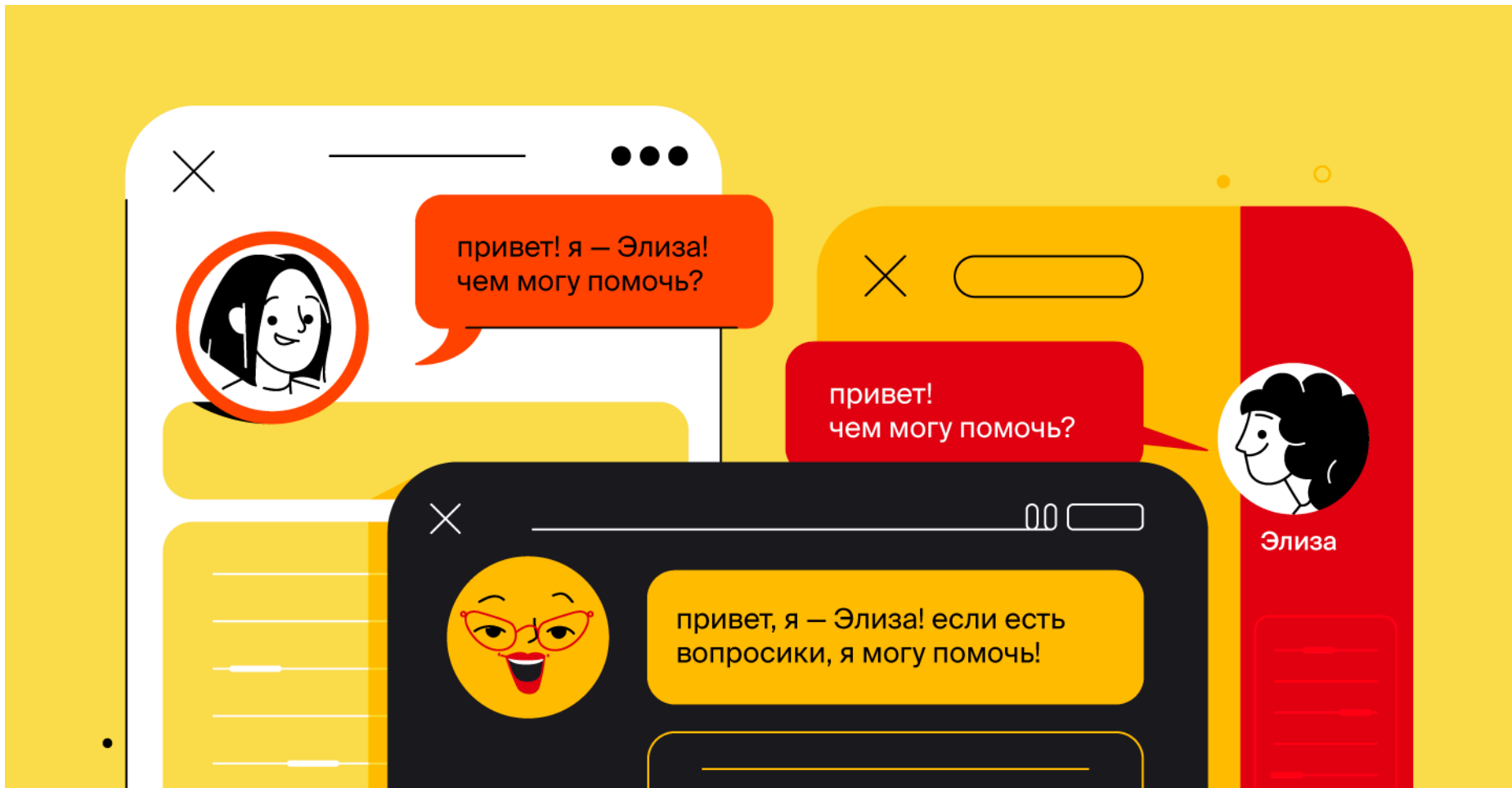
```
public class Bot {  
    String name;  
  
    public Bot() { // конструктор без параметров  
        name = "Элиза"; // проинициализировали поле  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        Bot bot = new Bot();  
        System.out.println("Вас приветствует чат-бот " + bot.name);  
    }  
}
```



Результат

Вас приветствует чат-бот Элиза

В этом случае у всех объектов-ботов тоже будет одинаковое значение поля **name** — «Элиза».



Удобнее иметь возможность создавать объекты с разными значениями полей. Для этого нужен конструктор с параметрами:

```
public class Bot {  
    String name;  
  
    public Bot(String newName) { // конструктор с параметром  
        name = newName;  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        Bot bot = new Bot("Томирис"); // можно передать любое имя  
        System.out.println("Вас приветствует чат-бот " + bot.name);  
    }  
}
```


Зачем this

Конструктор с параметрами работает так: сохраняет переданное значение в поле класса. Поэтому параметры конструктора удобнее всего называть так же, как и поля. Однако если это сделать — произойдёт ошибка:

```
public class Bot {
    String name;

    public Bot(String name) {
        name = name; // здесь ошибка, поле останется непроинициализировано
    }
}

public class Practice {
    public static void main(String[] args) {
        Bot bot = new Bot("Томирис");
        System.out.println("Вас приветствует чат-бот " + bot.name);
    }
}
```



Результат

Вас приветствует чат-бот null



Зачем `this`

Когда имена поля и параметра полностью совпадают — непонятно, что именно и какой переменной присваивать. Один из вариантов, как избежать этой ошибки, — использовать похожие имена. Например, такие, как **name** и **newName**. Именно так мы создавали конструкторы с параметрами до этого момента.

Однако есть более удобный способ — при помощи ключевого слова **this** (англ. «этот»):

```
public class Bot {
    String name;

    public Bot(String name) { // конструктор с параметром
        this.name = name; // используем this
    }
}

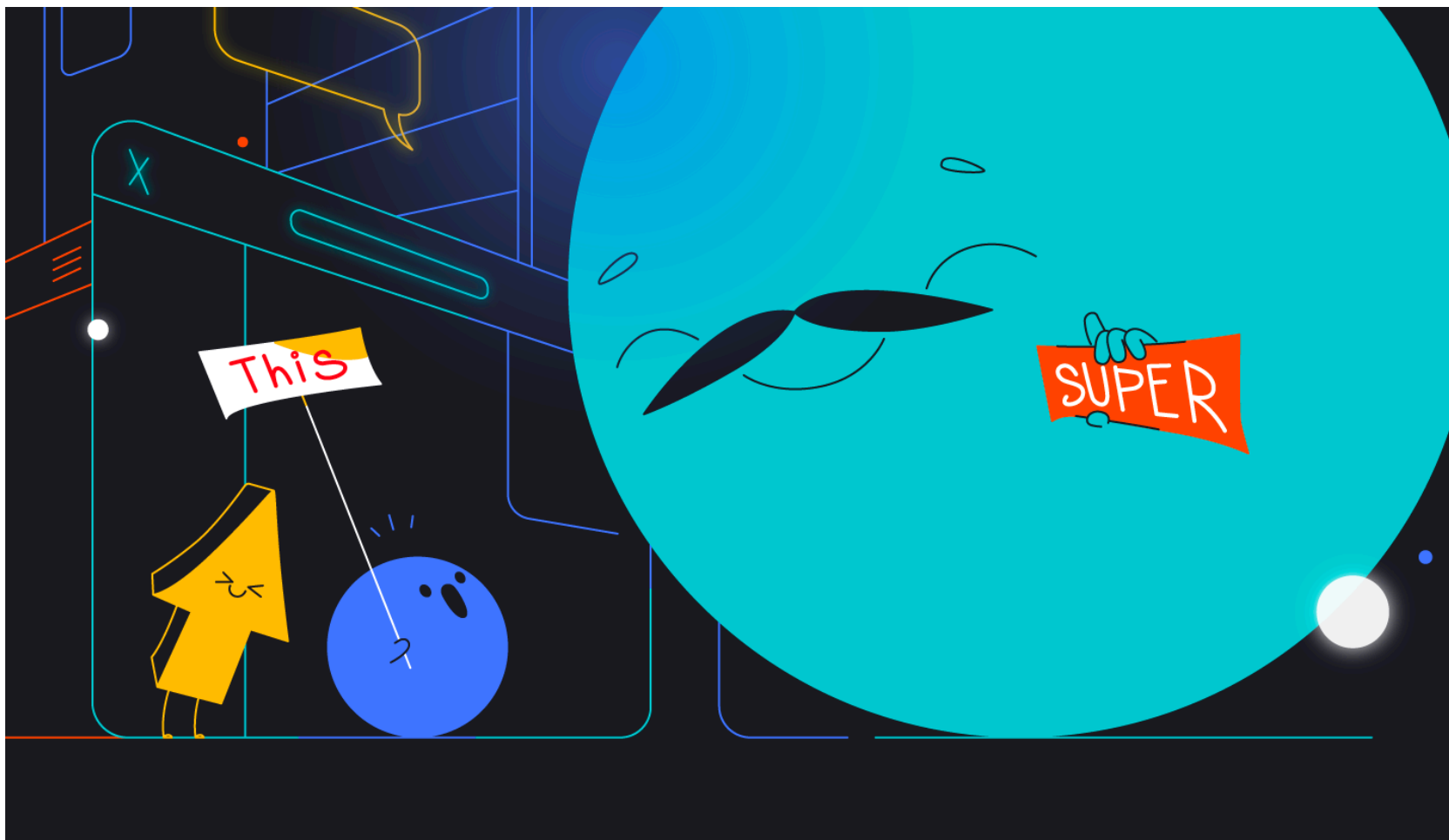
public class Practice {
    public static void main(String[] args) {
        Bot bot = new Bot("Алекс");
        System.out.println("Вас приветствует чат-бот " + bot.name);
    }
}
```



Результат

Вас приветствует чат-бот Алекс

Слово **this** — это ссылка на объект текущего класса. Оно позволяет обратиться к полю, методу или конструктору класса внутри него самого. Обращаться к полям в конструкторе через **this** — общепринятое соглашение. Это исключает путаницу с именами — сразу понятно, что значение аргумента **name**, переданное в конструктор, нужно присвоить полю **name** текущего (**this**) класса — **Bot**.



Определите, какой результат будет сохранён в переменной **result**.

```
public class Java {  
    int variable = 1;  
}  
  
public class Variable extends Java {  
    int variable = 2;  
  
    public int count() {  
        int variable = 3;  
        variable = super.variable;  
        return (this.variable + variable);  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        Variable variable = new Variable();  
        int result = variable.count();  
    }  
}
```



Ответ

3

Внутри метода `count()` значение его переменной `variable` (3) было перезаписано на значение поля `variable` из класса `Java` — 1. Затем к единице прибавилось значение поля `variable(2)` из её класса `Variable`.



Оптимизируем код

В теле конструктора не только инициализируются значения полей. В нём могут выполняться какие-то действия, например, подсчёт количества объектов или печать. Через **this** так же, как и через **super**, можно обратиться к другим конструкторам, чтобы не дублировать код:

Напомним, в классе может быть несколько конструкторов, главное, чтобы списки их параметров отличались.

```
public class Bot {
    String name;

    public Bot() { // конструктор без параметров
        System.out.println("Чат-бот создан");
    }

    public Bot(String name) { // конструктор с параметром
        this(); // вызвали конструктор без параметров
        this.name = name;
        System.out.println("Вас приветствует " + name);
    }
}

public class Practice {
    public static void main(String[] args) {
        Bot bot = new Bot("R2-D2");
    }
}
```



Результат

```
Чат-бот создан  
Вас приветствует R2-D2
```

Вызов конструктора через **this** должен быть на первом месте или сразу после вызова конструктора родительского класса через **super**.

Ещё один способ оптимизировать код — передавать в конструктор не поля по отдельности, а сразу объект:

```
public class Bot {
    String name;
    String specialization;

    public Bot(String name, String specialization) {
        this.name = name;
        this.specialization = specialization;
        System.out.println("Чат-бот " + name + " создан");
        System.out.println("Категория бота: " + specialization);
    }
}

public class Conversation {
    String botName;
    String botSpecialization;
    String greeting;

    public Conversation(Bot bot, String greeting) { // конструктор принимает объект
        this.botName = bot.name; // значения нужно сохранить в полях объекта-параметра
        this.botSpecialization = bot.specialization;
        this.greeting = greeting;
        System.out.println(greeting + ", " + botName + "!");
        System.out.println(botSpecialization + " - отличная тема для разговора!");
    }
}
```

```
public class Practice {  
    public static void main(String[] args) {  
        Bot bot = new Bot("С-3Р0", "Этикет, обычаи и переводы");  
        Conversation conversation = new Conversation(bot, "Привет");  
    }  
}
```




Результат

Чат-бот C-3P0 создан

Категория бота: Этикет, обычаи и переводы

Привет, C-3P0!

Этикет, обычаи и переводы - отличная тема для разговора!



Когда параметр — объект, то внутри конструктора нужно обратиться к его полям через их названия и точечную нотацию. Точно также в конструктор подкласса можно передавать объект класса-родителя.

```
public class Bot {  
    String name;  
    String specialization;  
  
    public Bot(String name, String specialization) {  
        this.name = name;  
        this.specialization = specialization;  
    }  
}
```

```
public class ChannelBot extends Bot {
    String channel;

    // конструктор принимает объект суперкласса
    public ChannelBot(Bot bot, String channel) {
        super(bot.name, bot.specialization);
        this.channel = channel;
        System.out.println("Привет! Я - " + name + "!");
        System.out.println("Я специалист по теме " + specialization + "!");
        System.out.println("Рад приветствовать тебя на нашем канале «" + channel + "»");
    }
}

public class Practice {
    public static void main(String[] args) {
        Bot bot = new Bot("Тарантино", "кино");
        ChannelBot channelBot = new ChannelBot(bot, "Опять смотреть нечего!");
    }
}
```




Результат

Привет! Я - Тарантино!

Я специалист по теме кино!

Рад приветствовать тебя на нашем канале «Опять смотреть нечего!»