




План занятия

1. Класс **Object** - австралопитек в программировании
2. Сравниваем объекты с помощью **equals(Object)**
3. Вычисляем хеш-код через **hashCode()**



Класс `Object` — австралопитек в программировании

Класс `Object` — австралопитек в программировании

У всех классов в Java есть общий предок — класс `Object`. Он входит в пакет `java.lang` и открывает доступ к полезному набору методов — их можно переопределить для целей вашей программы.



Класс `Object` — австралопитек в программировании

Наследование от класса `Object` происходит по умолчанию. Расширение при помощи `extends` не требуется. То есть обычное объявление класса, например `class Bird { ... }`, равнозначно такому — `class Bird extends Object { ... }`.

```
public class Bird /*extends Object*/ {  
    public void fly() { ... }  
}
```

Класс `Object` — австралопитек в программировании

В Java напрямую можно наследовать только от одного класса. Если у класса `Bird` появится наследник `Sparrow` — он автоматически унаследует `Object`:

```
public class Sparrow extends Bird { // Sparrow и Bird наследники Object
    public void tweet() { ... }
}
```

Правильные утверждения о классе Object

- A. Object - наследник любого класса в Java
- B. `private class Manager extends Person` — класс Manager является наследником Object.
- C. `public class Manager extends Person` — класс Person является наследником Object, а Manager — нет
- D. Любой класс в Java является наследником класса Object

Правильные ответы

A. Object - наследник любого класса в Java
Object — общий предок всех классов в Java.

B. `private class Manager extends Person` — класс Manager является наследником Object.

Любой класс является наследником Object. При этом неважно, какие у класса модификаторы доступа.

C. `public class Manager extends Person` — класс Person является наследником Object, а Manager — нет
Оба класса — наследники Object.

D. Любой класс в Java является наследником класса Object
Наследование от Object происходит, даже если не написано, что нужно явно его расширять при помощи ключевого слова `extends`.



Object как тип данных

Переменной типа **Object** можно присвоить любое ссылочное значение. Это может быть объект любого класса, например, список, массив или ваш собственный, а также значение примитивного типа после автоупаковки в класс-обёртку:


```

// Object не нужно импортировать явно, так как он входит в пакет java.lang

public class Practice {
    public static void main(String[] args) {
        Object anyObject; // переменная типа Object может принимать любой объект

        anyObject = new Person("Евламий"); // например, объект любого класса
        System.out.println(anyObject);

        // массив объектов
        Person[] people = { new Person("Агафья"), new Person("Пантелеймон") };
        anyObject = people;
        System.out.println(anyObject);

        // массив со значениями примитивного типа
        anyObject = new int[]{ 1, 2, 3, 4 };
        System.out.println(anyObject);

        anyObject = 42; // благодаря автоупаковке в Integer, так тоже можно
        System.out.println(anyObject);
        anyObject = true; // тут произойдёт автоупаковка в Boolean
        System.out.println(anyObject);

        // строки - это объекты класса String
        anyObject = "Вот так тоже можно";
        System.out.println(anyObject);
    }
}

public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
}

```

Результат

```
Person@36baf30c  
[LPerson;@7a81197d  
[I@5ca881b5  
42  
true  
Вот так тоже можно
```

Несмотря на то, что мы присваиваем значения разных типов — код компилируется без ошибок.



Object как тип данных

Благодаря тому, что **Object** может хранить любой объект, его удобно использовать как параметр универсального метода, который должен принимать объекты разных классов. Это можно увидеть в некоторых классах стандартной библиотеки, например, в методах хеш-таблиц, таких как **get(Object key)**, **remove(Object key)** или **containsValue(Object value)**, где в качестве ключа и значения могут быть любые объекты.



Object как тип данных

Метод с **Object** в сигнатуре можно написать и самостоятельно. Например, для подсчёта количества элементов в массивах (массив — это всегда объект, вне зависимости от типа его элементов) создадим метод **sizeof(Object[])**:

```

public class Practice {
    public static void main(String[] args) {
        Person[] people = { new Person("Томирис"), new Person("Дамир") };
        int peopleCount = sizeof(people);
        System.out.println("В массиве people " + peopleCount + " элемента.");

        String[] names = { "Максим", "Абай", "Тамерлан", "Александр"};
        int namesCount = sizeof(names);
        System.out.println("В массиве names " + namesCount + " элемента.");

        Integer[] numbers = { 42, 24, 45, 34, 23, 43, 54, 65, 43 };
        int numbersCount = sizeof(numbers);
        System.out.println("В массиве numbers " + numbersCount + " элемента.");
    }

    public static int sizeof(Object[] array) { // параметр - массив элементов типа Object
        int count = 0;
        for (Object o : array) {
            count++;
        }
        return count;
    }
}

public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }
}

```



Результат

В массиве `people` 2 элемента.
В массиве `names` 4 элемента.
В массиве `numbers` 9 элементов.



Object как тип данных

В качестве параметра у метода `sizeof()` указан массив объектов — `Object[] array`. Это делает его универсальным — позволяет работать с любыми массивами. Вы можете это проверить — добавить массив с элементами другого типа и посчитать в нём количество элементов.

Методы `Object`

Поскольку все классы наследуют поведение `Object`, Java-разработчику важно знать, как можно использовать его методы. Актуальную информацию о методах любых классов стандартной библиотеки всегда можно найти в официальной [документации Oracle](#). В 11-й версии Java у `Object` десять методов:

Начиная с 9-й версии Java, 11-й метод `finalize()` не используется при написании новых программ и помечен как устаревший

● Метод

● Описание

`public boolean equals (Object)`

Возвращает результат проверки объектов на равенство

`public int hashCode()`

Возвращает хеш-код объекта

`public String toString()`

Возвращает строковое представление объекта

`public final Class<?> getClass()`

Возвращает класс объекта

`protected Object clone()`

Возвращает копию объекта

`public final void wait()`

Освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока его работу не прервут или не известят об окончании ожидания

`public final void wait
(long timeoutMillis)`

То же самое, что и `wait()`, но можно указать предел времени ожидания в миллисекундах, по истечении которого поток возобновляет работу

`public final wait
(long timeoutMillis, int nanos)`

То же самое, что и `wait(long timeoutMillis)`, но дополнительно можно указать наносекунды

`public final void notify()`

Возобновляет работу одного из потоков, которые ранее вызывали метод `wait`

`public final void notifyAll()`

Возобновляет работу всех потоков, вызывавших ранее метод `wait`



Методы `Object`

Все варианты метода `wait()`, а также методы `notify()` и `notifyAll()` связаны с параллельным выполнением потоков. Это отдельная сложная тема — вы изучите её в другом модуле курса. Сейчас нужно запомнить, что эти пять методов относятся к **final** методам — их нельзя переопределить.



Методы Object

Метод `clone()` нужен, чтобы создавать копии текущего объекта. Его сложно переопределять и он редко используется — объекты удобнее копировать при помощи конструктора. Про `clone()` могут спросить на собеседовании — поэтому мы чуть более подробно остановимся на нём в модуле, посвящённом трудоустройству.




Методы `Object`

В этой теме подробно обсудим методы, которые разработчик может переопределять и активно использовать в своих классах. Это методы `equals(Object)`, `hashCode()` и `toString()`. Также разберём их контракты — правила и характеристики в документации, которым должны соответствовать переопределённые методы. Затронем метод `getClass()` — он нужен при переопределении `equals(Object)`.



Какие утверждения про класс `Object` правильные?

- A) Переменная типа `Object` может хранить только объекты классов-обёрток: `Integer`, `Double`, `Long`.
- B) В метод с параметром типа `Object` в качестве аргумента можно передать ссылку на объект любого класса.
- C) У `Object` много методов, например `isEqual(Object)`, `doDescription()` и `getCode()`.
- D) В переменную `Object` не получится сохранить массив.



A. Переменная типа **Object** может хранить только объекты классов-обёрток: **Integer**, **Double**, **Long**.

Переменная класса **Object** может принимать объект любого класса.

B. В метод с параметром типа **Object** в качестве аргумента можно передать ссылку на объект любого класса.

Так как **Object** родитель всех классов в Java, то его переменные могут хранить любые объекты.

C. У **Object** много методов, например **isEqual(Object)**, **doDescription()** и **getCode()**.
Таких методов в **Object** нет. Зато есть
методы **equals(Object)**, **toString()** и **hashCode()** — их и разберём подробно.

D. В переменную **Object** не получится сохранить массив.
Массивы — это объекты вне зависимости от типа их элементов.




Задача

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



Решение

<https://gist.github.com/practicetasks/f89c74a41e3b163a1b1d88d17be98ef5>



**Сравниваем объекты с помощью
`equals(Object)`**



Сравниваем объекты с помощью `equals(Object)`

Разбор методов `Object` начнём с `equals(Object)` (от англ. *equal* — «равняться, быть равным»). Этот метод проверяет объекты на равенство и возвращает результат в виде булева значения — `true`, если объекты равны, или `false` — если нет.



Почему не хватит ==

Метод `equals(Object)` по сути аналогичен оператору `==`. Разница в том, что при помощи `==` сравниваются значения примитивных типов, а при помощи `equals(Object)` — ссылочных. Для проверки объектов на равенство `==` не подходит.

При помощи оператора `==` можно сравнить только ссылки — буклеты, а не содержание — саму технику. Из-за этого, даже если объекты будут идентичны, их сравнение при помощи `==` даст неверный результат.

```
public class Book {  
    public String title;  
    public String author;  
    public int pageNumber;  
  
    public Book(String title, String author, int pageNumber) {  
        this.title = title;  
        this.author = author;  
        this.pageNumber = pageNumber;  
    }  
}
```

```
public class Practice {  
    public static void main(String[] args) {  
        // сохраняем одно и то же число в две переменные  
        int variable1 = 42;  
        int variable2 = 42;  
  
        boolean result = variable1 == variable2; // сравниваем значения переменных  
        System.out.println(result); // значения равны  
  
        String title = "Java для начинающих";  
        String author = "Ансар Сеньёров";  
        int pageNumber = 777;  
  
        // передаём одни и те же данные двум объектам  
        Book book1 = new Book(title, author, pageNumber);  
        Book book2 = new Book(title, author, pageNumber);  
  
        boolean result2 = book1 == book2; // сравниваем значения объектных переменных  
        System.out.println(result2); // получили некорректный результат  
    }  
}
```



Результат

```
true  
false
```

Несмотря на то, что у книг-объектов одинаковые названия и один и тот же автор и число страниц, сравнение оператором `==` отрицает их равенство. Всё потому, что переменные **book1** и **book2** содержат разные ссылки. А нам бы хотелось, чтобы программа могла определить, что если атрибуты книги — автор, название и число страниц — совпадают, значит, это одна и та же книга.

Примитивные
переменные

variable 1

42

variable 2

Объектные
переменные

book1 = new Book(...);

7334aada

title = "..."
author = "..."

book2 = new Book(...);

1d9b7cce

title = "..."
author = "..."

`!=`

Ссылки — только один из аспектов, по которому можно сравнить объекты между собой. Поэтому оператора `==` всегда будет недостаточно. Чтобы получить корректный результат, нужен метод `equals(Object)`.



Где используется правильный способ проверки на равенство:

A)

```
int x = 10;  
int y = x++;  
  
boolean result = (x == y);
```

B)

```
long a = 1_000_000_000_000L;  
long b = 2_000_000_000_000L;  
long c = b - a;  
  
boolean result = (a.equals(c));
```

C)

```
Book b1 = new Book();  
Book b2 = new Book();  
  
boolean result = (b1.equals(b2));
```

D)

```
boolean result = ("Java" == "Java");
```

ОТВЕТ

A)

```
int x = 10;  
int y = x++;  
  
boolean result = (x == y);
```

C)

```
Book b1 = new Book();  
Book b2 = new Book();  
  
boolean result = (b1.equals(b2));
```

Переопределение equals

Чтобы сравнить объекты через метод `equals(Object)`, нужно его переопределить. В базовой реализации класса `Object` метод выглядит так:

```
// реализация по умолчанию  
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Здесь сравниваются адреса объектов через `==`. Это сделано так, потому что заранее неизвестно, какие будут объекты и какие их атрибуты нужно будет сравнить.

Переопределение equals

При этом в любом случае сравнение начнётся с сопоставления ссылок. Если ссылки совпадают — это один и тот же объект, и проводить более углублённое сравнение нет необходимости.

Во многих классах стандартной библиотеки метод `equals(Object)` уже переопределён. Это даёт возможность применять его автоматически. Например, в классе `String` можно применять `equals(Object)` для сравнения строковых переменных без дополнительных действий:

Для сравнения объектов собственных классов, будь то книги, хомяки или космические корабли, метод `equals(Object)` нужно переопределить. Начинаем с аннотации `@Override` и стандартной реализации метода:

```
public class Book {
    public String title;
    public String author;
    public int pageNumber;

    public Book(String title, String author, int pageNumber) {
        this.title = title;
        this.author = author;
        this.pageNumber = pageNumber;
    }

    @Override // аннотация сигнализирует о том, что мы переопределяем метод
    public boolean equals(Object obj) {
        if (this == obj) return true; // проверяем адреса объектов
    }
}
```



Переопределение equals

Так вы сразу вычислите, не имеете ли дело с одним и тем же объектом. Если это так, то нет смысла дальше проверять все поля на равенство, можно сразу вернуть положительный результат.

Следующим шагом нужно проверить, не была ли передана в метод `equals(Object)` пустая ссылка `null` вместо объекта. Если аргумент равен `null` — можно сразу возвращать отрицательный результат:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true; // проверяем адреса объектов
    if (obj == null) return false; // проверяем ссылку на null
}
```

Если вовремя не отловить `null` и продолжать дальнейшую проверку пустой ссылки, это приведёт к генерации исключения `NullPointerException`.

Поскольку базово метод `equals(Object)` принимает в качестве аргумента объекты любых классов, дальше требуется проверить, что в него передан экземпляр нужного. В примере это класс **Book**. Провести такую проверку поможет другой метод класса **Object** — `getClass()`. Этот метод возвращает информацию о том, к какому классу относится объект. Если классы у сравниваемых объектов отличаются, то вернётся **false**:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true; // проверяем адреса объектов
    if (obj == null) return false; // проверяем ссылку на null
    if (this.getClass() != obj.getClass()) return false; // сравниваем классы объектов
}
```


Первая часть переопределения метода завершена — исключено, что это один и тот же объект, экземпляры разных классов или передана пустая ссылка. Эти проверки нужно провести вне зависимости от того, объекты каких классов вы сравниваете между собой.

Далее нужно привести переданный объект к тому классу, где переопределяется `equals(Object)`. Приведение любых типов, в том числе ссылочных, осуществляется с помощью круглых скобок:

```
@Override
```

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // проверяем адреса объектов  
    if (obj == null) return false; // проверяем ссылку на null  
    if (this.getClass() != obj.getClass()) return false; // сравниваем классы объектов  
    Book otherBook = (Book) obj; // привели второй объект к классу Book  
}
```

Приведение типов нужно, чтобы получить доступ к полям второго объекта. После этого можно обращаться к ним по выбранному имени (**otherBook**), используя точечную нотацию.

Вторая часть переопределения метода **equals(Object)** касается сравнения полей объектов. В классе **Book** три поля **title** — название, **author** — автор и **pageNumber** — количество страниц. Нужно проверить, что их значения совпадают и что объектные поля не содержат пустую ссылку **null** (исключена ошибка **NullPointerException**).

Для сравнения полей удобно пользоваться методом `Objects.equals(Object, Object)`.
Утилитарный класс `Objects` (с `s` на конце — подробнее о нём можно почитать [здесь](#))
содержит набор вспомогательных методов, в том числе `equals(Object, Object)`.

Этот метод сначала проверяет, не равны ли переданные аргументы пустым ссылкам, и если нет — сравнивает их. Поля примитивных типов сравниваем через оператор `==`:

```
import java.util.Objects; // импортируем класс Objects

public class Book {
    public String title;
    public String author;
    public int pageNumber;

    public Book(String title, String author, int pageNumber) {
        this.title = title;
        this.author = author;
        this.pageNumber = pageNumber;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // проверяем адреса объектов
        if (obj == null) return false; // проверяем ссылку на null
        if (this.getClass() != obj.getClass()) return false; // сравниваем классы
        Book otherBook = (Book) obj; // открываем доступ к полям другого объекта
        return Objects.equals(title, otherBook.title) && // проверяем все поля
            Objects.equals(author, otherBook.author) && // нужно логическое «и»
            (pageNumber == otherBook.pageNumber); // примитивы сравниваем через ==
    }
}
```



Проверка после переопределения

Переопределить метод `equals(Object)` не так уж и просто. Поэтому в некоторых случаях это не требуется — например, когда класс несёт сервисную или утилитарную функциональность или был создан только ради использования его методов. Также можно унаследовать переопределённый `equals(Object)` с подходящей реализацией.

Если всё-таки требуется написать новую реализацию `equals(Object)`, то она должна соответствовать контракту метода — своду правил, закреплённых в документации.

Разберём их:

Правило рефлексивности — объект должен быть равен самому себе. То есть вызов `x.equals(x)` должен всегда возвращать **true**.

Правило симметричности — «от перестановки мест слагаемых сумма не меняется». Результат сравнения объектов не зависит от того, в каком порядке они расположены. Вызов `x.equals(y)` должен возвращать **true** в то же время, когда вызов `y.equals(x)` возвращает **true**.

Правило логической транзитивности — если два объекта равны и один из них равен третьему, то все три объекта равны. Так, если вызов `x.equals(y)` возвращает **true** и `y.equals(z)` возвращает **true**, то вызов `x.equals(z)` также должен вернуть **true**.

Правило согласованности — если не менять данные сравниваемых объектов, то и результат их сравнения должен быть всегда одинаков. То есть множественный вызов `x.equals(y)` должен возвращать один и тот же результат до тех пор, пока данные полей объектов `x` и `y` неизменны.

Правило «на ноль делить нельзя» — ни один из сравниваемых объектов не может быть равен `null`. Это значит, что вызов `x.equals(null)` должен всегда возвращать `false`.



Задача

Подставьте код из этой ссылки и проверьте работоспособность метода **equals()**
<https://gist.github.com/practicetasks/8d5ff11f5a023008e3d269ab820fad99>


```
public class Song {  
    public String title;  
    public String artist;  
    public String songwriter;  
  
    public Song(String title, String artist, String songwriter) {  
        this.title = title;  
        this.artist = artist;  
        this.songwriter = songwriter;  
    }  
  
    // переопределите метод equals(Object)  
    ...  
}
```

Решение

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Song song = (Song) o;
    return Objects.equals(title, song.title)
        && Objects.equals(artist, song.artist)
        && Objects.equals(songwriter, song.songwriter);
}
```



**Вычисляем хеш-код через
hashCode()**



Вычисляем хеш-код через `hashCode()`

Вместе с методом `equals(Object)` сразу стоит переопределить другой метод `Object` — `hashCode()`. Он оптимизирует хранение и поиск объектов в коллекциях, таких как `HashMap` и других. В этом уроке разберём, как этот метод работает, как его переопределять и почему он идёт в связке с `equals(Object)`.



Хеширование и хеш

Каждый объект в программе можно представить в виде некоторого целого числа. Процесс вычисления такого числа называется **хешированием**, а его результат — **хешем**. Этим как раз и занимается метод `hashCode()` — генерирует хеш для объектов, чтобы их легче было сортировать и искать.

Хеширование и хеш

Разберём на примере. Представьте, у вас есть документ, и нужно найти его копию в большой стопке бумаг на столе. Чтобы это сделать, нужно поочерёдно изучить все бумаги. Другое дело, если документы рассортированы по папкам с серийным номером. Поиск упрощается — по номеру легко найти нужную папку, а потом взять оттуда копию. Проверять все папки и документы — не нужно. Серийный номер стопки — это и есть хеш-код, сгенерированный методом `hashCode()`.



Переопределение hashCode()

Базовая реализация метода `hashCode()` стремится создать уникальный хеш для каждого объекта, в том числе для идентичных. При переопределении нужно это исправить.

💡 У всех стандартных ссылочных типов данных в Java (**String**, **Integer**, **Double** и т. д.) методы `equals(Object)` и `hashCode()` уже корректно переопределены. Поэтому их можно спокойно использовать с коллекциями **HashMap**, **HashSet** и прочими.



Переопределение hashCode()

Чтобы хеш-коды разных объектов отличались, а одинаковых — совпадали, нужно вычислять хеш в связке с методом **equals(Object)**. Оба метода должны зависеть от одних и тех же полей. Отсюда и взялось правило, что при переопределении **equals(Object)** лучше сразу переопределять метод **hashCode()**.



Переопределение hashCode()

Более простой и самый распространённый вариант переопределения **hashCode()** — через метод **hash(Object... values)** уже знакомого вам класса **Objects**. Этот метод генерирует хеш-код для последовательности переданных в него значений. Реализация с его использованием лаконична — вызываем метод и передаём в него нужные поля (те же, что и в методе **equals(Object)**):

```
import java.util.Objects;

public class Person {
    public String firstName;
    public String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(firstName, person.firstName) &&
            Objects.equals(lastName, person.lastName);
    }

    @Override
    public int hashCode() {
        // вызываем вспомогательный метод и передаём в него нужные поля
        return Objects.hash(firstName, lastName);
    }
}
```

Сверяемся с контрактом

У метода `hashCode()` есть [контракт](#), которым нужно руководствоваться при его ручном переопределении. Он включает три правила:

- Если при сравнении методом `equals(Object)` объекты оказались равны, то `hashCode()` должен возвращать у каждого из них одно и то же число.
- Метод `hashCode()` должен возвращать одно и то же целое число до тех пор, пока значения полей, используемых в методе `equals(Object)` того же класса, остаются прежними.
- Нужно стремиться к тому, чтобы у объектов, которые не равны при сравнении `equals(Object)`, были разные хеш-коды, но учитывать, что они могут совпасть. Поэтому, если у двух объектов одинаковые хеш-коды, нельзя утверждать, что объекты равны. Точный результат покажет только метод `equals(Object)`.



Сверяемся с контрактом

Если подытожить, переопределяя метод `hashCode()`, важно проверить, чтобы для равных объектов всегда возвращался одинаковый хеш-код, а для разных по возможности разные.



Пара `equals(Object)`-`hashCode()` и поиск в хеш-таблицах

Правильно реализованная пара `equals(Object)` и `hashCode()` делает возможным поиск объектов в списках и хеш-таблицах. К примеру, создадим список людей `persons` и таблицу для хранения их телефонных номеров `contacts`. Ключом в таблице будет объект `Person`, а значением — номер телефона. Сейчас поиск элементов ни в списке, ни в хеш-таблице невозможен:

```
public class Person {  
    public String firstName;  
    public String lastName;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

```

import java.util.ArrayList;
import java.util.HashMap;

public class Practice {
    public static void main(String[] args) {
        ArrayList<Person> persons = new ArrayList<>(); // список имён и фамилий
        HashMap<Person, String> contacts = new HashMap<>(); // хеш-таблица контактов

        String firstName = "Стив";
        String lastName = "Джобс";
        String phoneNumber = "8 (777) 123-45-67";

        persons.add(new Person(firstName, lastName)); // добавляем элемент в список

        contacts.put(new Person(firstName, lastName), phoneNumber); // добавляем элемент в таблицу

        System.out.println("Количество людей в списке: " + persons.size() +
            ", контактов: " + contacts.size()); // проверяем наличие элементов

        if (persons.contains(new Person(firstName, lastName))) { // ищем элемент в списке
            System.out.println("Человек с именем " + firstName +
                " и фамилией " + lastName + " найден в списке.");
        } else {
            System.out.println("Метод equals у класса Person реализован неверно!");
        }

        // ищем элемент в таблице по ключу:
        if (contacts.containsKey(new Person(firstName, lastName))) {
            System.out.println("Человек с именем " + firstName + " и фамилией " +
                lastName + " найден в таблице контактов. Его телефонный номер: " +
                contacts.get(new Person(firstName, lastName)));
        } else {
            System.out.println("Метод hashCode у класса Person реализован неверно!");
        }
    }
}

```

Результат

```
Количество людей в списке: 1, контактов: 1  
Метод equals у класса Person реализован неверно!  
Метод hashCode у класса Person реализован неверно!
```

Несмотря на то, что элементы добавлены — методы поиска в списках **contains()** и в таблицах **containsKey()** не могут их найти.

Без переопределённых методов `equals(Object)` и `hashCode()` программа превратилась в того самого растерянного человека, который пытается найти копию документа на столе, заваленном бумагами. Поможем ему — переопределим метод `equals(Object)`. С его помощью метод `contains(Object)` класса `ArrayList` сможет один за другим сверить каждый элемент списка с искомым:

```
import java.util.Objects;

public class Person {
    public String firstName;
    public String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public boolean equals(Object o) { // добавили и переопределили equals
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(firstName, person.firstName) &&
            Objects.equals(lastName, person.lastName);
    }
}
```

```

import java.util.ArrayList;
import java.util.HashMap;

public class Practice {
    public static void main(String[] args) {
        ArrayList<Person> persons = new ArrayList<>(); // список имён и фамилий
        HashMap<Person, String> contacts = new HashMap<>(); // хеш-таблица контактов

        String firstName = "Стив";
        String lastName = "Джобс";
        String phoneNumber = "8 (777) 123-45-67";

        persons.add(new Person(firstName, lastName)); // добавляем элемент в список

        contacts.put(new Person(firstName, lastName), phoneNumber); // добавляем элемент в таблицу

        System.out.println("Количество людей в списке: " + persons.size() +
            ", контактов: " + contacts.size()); // проверяем наличие элементов

        if (persons.contains(new Person(firstName, lastName))) { // ищем элемент в списке
            System.out.println("Человек с именем " + firstName +
                " и фамилией " + lastName + " найден в списке.");
        } else {
            System.out.println("Метод equals у класса Person реализован неверно!");
        }

        // ищем элемент в таблице по ключу:
        if (contacts.containsKey(new Person(firstName, lastName))) {
            System.out.println("Человек с именем " + firstName + " и фамилией " +
                lastName + " найден в таблице контактов. Его телефонный номер: " +
                contacts.get(new Person(firstName, lastName)));
        } else {
            System.out.println("Метод hashCode у класса Person реализован неверно!");
        }
    }
}

```



Результат

Количество людей в списке: 1, контактов: 1
Человек с именем Стив и фамилией Джобс найден в списке.
Метод hashCode у класса Person реализован неверно!

Если **equals(Object)** не переопределён — используется его базовая реализация, которая сравнивает только ссылки объектов и выдаёт некорректный результат. Теперь, когда **equals(Object)** работает корректно, в списке легко находится нужный объект.

Однако для поиска в хеш-таблице реализации только **equals(Object)** по-прежнему недостаточно. Пока не переопределён **hashCode()**, вызывается его базовая реализация — для одинаковых объектов возвращается разный хеш-код. В итоге метод **containsKey()** выдаёт неверный результат. Добавим в **Person** переопределённый **hashCode()**:

```

import java.util.Objects;

public class Person {
    public String firstName;
    public String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(firstName, person.firstName) &&
            Objects.equals(lastName, person.lastName);
    }

    @Override
    public int hashCode() {
        int hash = 17;
        if (firstName != null) {
            hash = hash + firstName.hashCode();
        }
        hash = hash * 31;

        if (lastName != null) {
            hash = hash + lastName.hashCode();
        }
        return hash;
    }
}

```

```

import java.util.ArrayList;
import java.util.HashMap;

public class Practice {
    public static void main(String[] args) {
        ArrayList<Person> persons = new ArrayList<>(); // список имён и фамилий
        HashMap<Person, String> contacts = new HashMap<>(); // хеш-таблица контактов

        String firstName = "Стив";
        String lastName = "Джобс";
        String phoneNumber = "8 (777) 123-45-67";

        persons.add(new Person(firstName, lastName)); // добавляем элемент в список

        contacts.put(new Person(firstName, lastName), phoneNumber); // добавляем элемент в таблицу

        System.out.println("Количество людей в списке: " + persons.size() +
            ", контактов: " + contacts.size()); // проверяем наличие элементов

        if (persons.contains(new Person(firstName, lastName))) { // ищем элемент в списке
            System.out.println("Человек с именем " + firstName +
                " и фамилией " + lastName + " найден в списке.");
        } else {
            System.out.println("Метод equals у класса Person реализован неверно!");
        }

        // ищем элемент в таблице по ключу:
        if (contacts.containsKey(new Person(firstName, lastName))) {
            System.out.println("Человек с именем " + firstName + " и фамилией " +
                lastName + " найден в таблице контактов. Его телефонный номер: " +
                contacts.get(new Person(firstName, lastName)));
        } else {
            System.out.println("Метод hashCode у класса Person реализован неверно!");
        }
    }
}

```



Результат

Количество людей в списке: 1, контактов: 1

Человек с именем Стив и фамилией Джобс найден в списке.

Человек с именем Стив и фамилией Джобс найден в таблице контактов. Его телефонный номер: 8 (777) 123-45-67

Теперь у равных объектов класса **Person** одинаковый хеш-код, поэтому получается найти в таблице нужную запись! В то же время благодаря корректной реализации у разных объектов будет разный хеш-код — что сделает поиск по таблице эффективным.

Какие утверждения верные?

- 1) Базовая реализация метода **hashCode()** всё время генерит одинаковый хеш объекта.
- 2) Базовая реализация метода **hashCode()** для каждого нового объекта стремится сгенерировать уникальное значение.
- 3) Метод **hashCode()** возвращает набор символов.
- 4) Для корректной работы хеш-таблицы достаточно переопределить только метод **hashCode**.
- 5) Хеш-код должен высчитываться на основе тех же полей, что используются для проверки равенства методом **equals(Object)**.
- 6) Стандартные ссылочные типы данных в Java, например, **Short**, **Long**, **String** — используют базовую реализацию **hashCode()**.

1) Базовая реализация метода **hashCode()** всё время генерит одинаковый хеш объекта. По умолчанию для каждого объекта вычисляется уникальный хеш-код.

2) Базовая реализация метода **hashCode()** для каждого нового объекта стремится сгенерировать уникальное значение.

Для генерации хеша может использоваться, например, адрес объекта или другие данные о нём.

3) Метод **hashCode()** возвращает набор символов.

Результатом хеширования может быть набор символов фиксированной длины. Но метод **hashCode()** класса **Object** возвращает целое число типа **int**.

4) Для корректной работы хеш-таблицы достаточно переопределить только метод **hashCode**.

Используя только **hashCode** нельзя утверждать, что объекты равны. Для этого также нужен правильно переопределённый метод **equals(Object)**.

5) Хеш-код должен высчитываться на основе тех же полей, что используются для проверки равенства методом **equals(Object)**.

Для равных объектов **hashCode()** должен возвращать одинаковое значение.

Поэтому **equals(Object)** и **hashCode()** должны использовать одни и те же поля.

6) Стандартные ссылочные типы данных в Java, например, **Short**, **Long**, **String** — используют базовую реализацию **hashCode()**.

Базовой реализации не хватило бы для корректной работы этих типов — во многих классах стандартной библиотеки **hashCode()** переопределён.




Задача

[https://github.com/practicetasks/java_tasks/blob/main/object_hashcode_equals/task_2/
README.md](https://github.com/practicetasks/java_tasks/blob/main/object_hashcode_equals/task_2/README.md)



Решение

<https://gist.github.com/practicetasks/8917cbc643468e44d6bc894289a4693d>

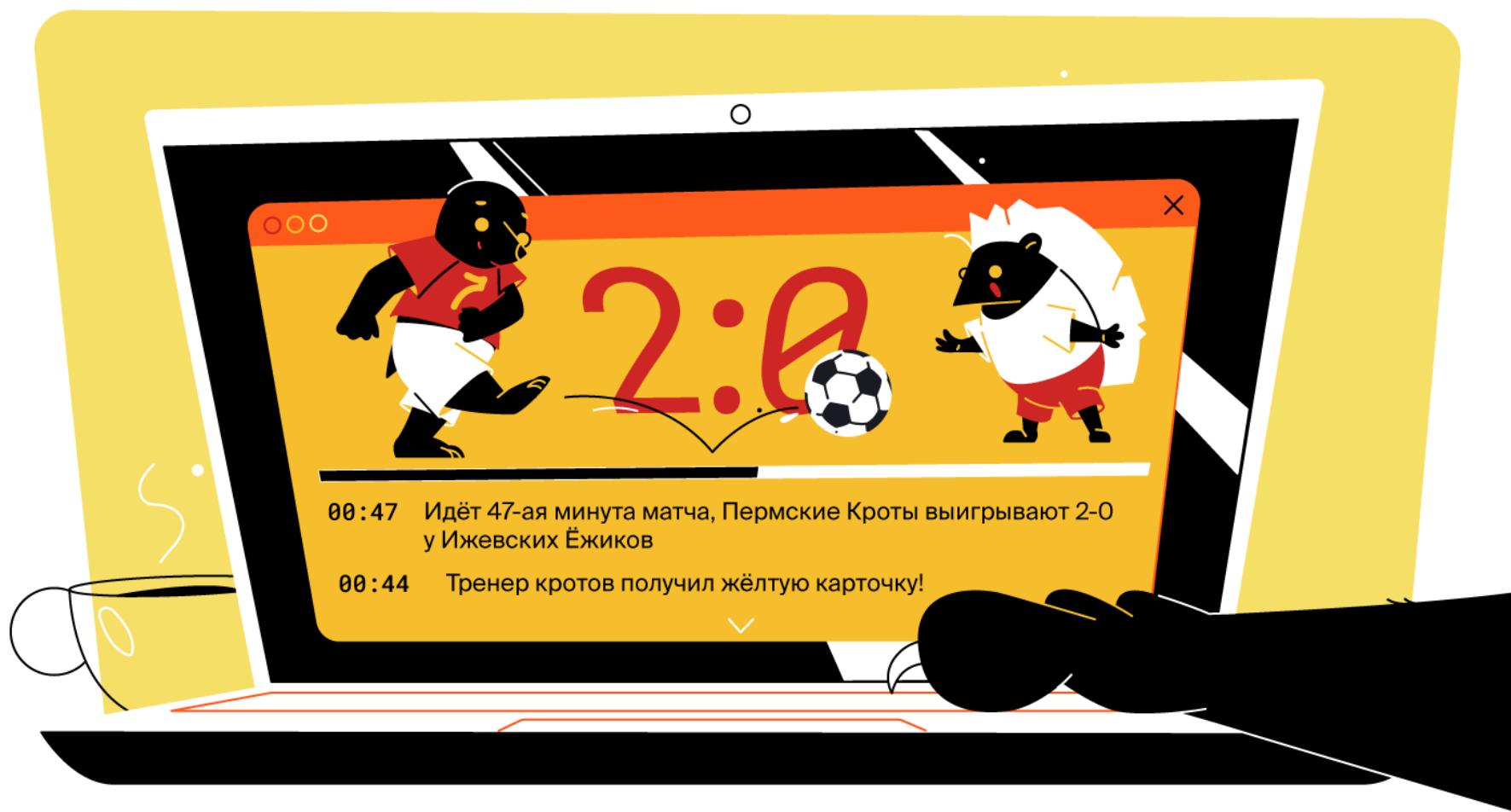


Заглядываем в объект с помощью
`toString()`

Ещё один метод класса **Object**, который разработчик может переопределять и активно использовать в своих классах — это **toString()**.

Контракт **toString()** не такой строгий, как у **equals(Object)** или **hashCode()**, и представляет собой описание работы метода. В нём указано, что **toString()** возвращает строку, которая представляет объект в виде текста. Полученное представление должно быть лаконичным и информативным.

Пока программа отлаживается в среде разработки, не всегда очевидно, зачем преобразовывать объект в текст. Но нужно учесть, что когда код будет запущен на рабочем сервере или попадёт на устройства пользователей, контролировать его работу станет сложнее. Чтобы это делать, важна возможность получить текущее состояние любого объекта в виде текста, что и делает метод `toString()`.



Так же, как `equals(Object)` и `hashCode()`, метод `toString()` рекомендуется всегда переопределять. Его базовая реализация в классе `Object` не информативна — не учитывает поля каждого класса. Она работает так:

```
public class Address {
    public String city; // город
    public String street; // улица
    public int houseNumber; // номер дома

    public Address(String city, String street, int houseNumber) {
        this.city = city;
        this.street = street;
        this.houseNumber = houseNumber;
    }
}

public class Practice {
    public static void main(String[] args) {
        Address address = new Address("Астана", "Туркестан", 34);

        System.out.println(address.toString()); // вызываем toString() с базовой реализацией
    }
}
```




Результат

Address@33c7353a

Сейчас `toString()` возвращает что-то вроде `Address@33c7353a`. Строка формируется из названия класса, символа-«собаки» `@` и хеш-кода объекта. Такое текстовое представление не поможет узнать состояние объекта или найти причину сбоя в программе. Кроме того, если вы создадите второй объект `Address` с теми же атрибутами, а затем выведете его в консоль через `toString()`, результат будет отличаться.



Переопределяем `toString()`

Цель переопределения `toString()` — получить простую по форме и ёмкую по содержанию информацию о содержании объекта. Реализация метода при этом зависит от разработчика. К примеру, можно переопределить `toString()` в классе **Address** так:

```

public class Address {
    public String city;
    public String street;
    public int houseNumber;

    public Address(String city, String street, int houseNumber) {
        this.city = city;
        this.street = street;
        this.houseNumber = houseNumber;
    }

    @Override // переопределяем toString
    public String toString() {
        return city + ", " + street + ", д. " + houseNumber; // просто возвращаем поля класса
    }
}

public class Practice {
    public static void main(String[] args) {
        Address address = new Address("Астана", "Туркестан", 34);
        Address address2 = new Address("Астана", "Улы Дала", 45);

        System.out.println("Адрес 1: " + address);
        System.out.println("Адрес 2: " + address2);
    }
}

```



Результат

Адрес 1: Астана, Туркестан, д. 34

Адрес 2: Астана, Улы Дала, д. 45

Результат получился гораздо информативнее, чем **Address@33c7353a**, однако всё ещё не включает важные детали, например, имя класса или названия полей.

Так как у `toString()` нет строгого контракта, при его переопределении принято руководствоваться вот такими рекомендациями:

1. Единый формат.

Когда вывод `toString()` построен во всех классах по одной и той же логике — код удобно читать и воспринимать. Детали могут отличаться, но основа должна быть единой. В начале указывается имя класса, затем в фигурных скобках названия полей и их значения:

```
@Override
public String toString() {
    return "Address{" + // имя класса
        "city='" + city + '\'' + // поле1=значение1
        ", street='" + street + '\'' + // поле2=значение2
        ", houseNumber=" + houseNumber + // поле3=значение3
        '}';
}
// будет напечатано: Address{city='Астана', street='Улы Дала', houseNumber=45}
```

Важно запомнить, что **toString()** не должен влиять на состояние объекта. Он работает в режиме «только чтение» — не меняет значения полей и не проводит с ними расчёты.

2. Лаконичность и информативность.

- В реализацию `toString()` стоит включать только те поля, которые содержат ключевую или определяющую информацию. Статические или вспомогательные поля можно опустить.
- Реализацию `toString()` важно поддерживать в актуальном состоянии. Добавлять поля или удалять их по мере необходимости.
- Некоторые поля могут содержать объёмные данные. Нет практического смысла в том, чтобы выводить их полное или даже сокращённое содержание. Можно отобразить их длину.

К примеру, добавим в **Address** поле **extraInfo**. В нём будет храниться дополнительная информация об адресах — историческая справка или архитектурный статус. Чтобы не выводить значение этого поля полностью, оставим в `toString()` только его размер:


```

public class Address {
    public String city;
    public String street;
    public int houseNumber;
    public String extraInfo;

    public Address(String city, String street, int houseNumber, String extraInfo) {
        this.city = city;
        this.street = street;
        this.houseNumber = houseNumber;
        this.extraInfo = extraInfo;
    }

    @Override
    public String toString() {
        return "Address{" +
            "city='" + city + '\'' +
            ", street='" + street + '\'' +
            ", houseNumber=" + houseNumber + '\'' +
            ", extraInfo.length=" + extraInfo.length() + // выводим не значение, а длину
            '}';
    }
}

```

Вывести длину вместо содержания также может быть удобно для полей-массивов или других коллекций.

3. Профилактика исключений `NullPointerException`.

Оператор конкатенации `+` умеет работать с пустыми ссылками `null`, поэтому при сложении строк с пустой ссылкой ошибки не будет. Исключение `NullPointerException` может возникнуть в том случае, если у одного из полей вызывается метод — в нашем примере у поля `extraInfo` перед конкатенацией вызывается метод `length`. Избежать ошибки можно так:

```

public class Address {
    public String city;
    public String street;
    public int houseNumber;
    public String extraInfo;

    public Address(String city, String street, int houseNumber, String extraInfo) {
        this.city = city;
        this.street = street;
        this.houseNumber = houseNumber;
        this.extraInfo = extraInfo;
    }

    @Override
    public String toString() {
        String result = "Address{" +
            "city='" + city + '\'' +
            ", street='" + street + '\'' +
            ", houseNumber=" + houseNumber + '\'';

        if (extraInfo != null) { // проверяем, что поле не содержит null
            result = result + ", extraInfo.length=" + extraInfo.length(); // выводим не значение, а длину
        } else {
            result = result + ", extraInfo=null"; // выводим информацию, что поле равно null
        }

        return result + '}';
    }
}

```

4. Форматирование данных.

Некоторые типы данных требуют дополнительного форматирования. Это актуально, к примеру, для массивов. Они наследуют базовую реализацию `toString()`, и чтобы посмотреть их содержание, лучше дополнительно вызвать метод `toString(Object[] a)` класса `Arrays`.. Этот метод проверяет массив на `null` и если всё в порядке, возвращает его текстовое представление.

```

import java.util.Arrays;

public class Address {
    public String city;
    public String street;
    public int houseNumber;
    public String extraInfo;
    public String[] residents;

    @Override
    public String toString() {
        String result = "Address{" +
            "city='" + city + '\'' +
            ", street='" + street + '\'' +
            ", houseNumber=" + houseNumber + '\'';

        if (extraInfo != null) { // проверяем, что поле не содержит null
            result = result + ", extraInfo.length=" + extraInfo.length(); // выводим не значение, а длину
        } else {
            result = result + ", extraInfo=null"; // выводим информацию, что поле равно null
        }

        return result +
            // форматируем массив с помощью метода Arrays.toString
            ", residents=" + Arrays.toString(residents) +
            '}';
    }
}

```

В отдельном форматировании могут также нуждаться такие данные, как даты (например, месяц указывать текстом или числом в формате 31.01.2021 или 2021-01-31), время (показывать ли миллисекунды — это может быть важно для банковских транзакций), валюта и другие. Если реализация **toString()** в переданном объекте не подходит, её можно и нужно адаптировать.

При соблюдении всех рекомендаций **toString()** вернёт простое и информативное представление объекта в текстовом виде:

```

import java.util.Arrays;

public class Address {
    public String city;
    public String street;
    public int houseNumber;
    public String extraInfo;
    public String[] residents;

    @Override
    public String toString() {
        String result = "Address{" +
            "city='" + city + '\'' +
            ", street='" + street + '\'' +
            ", houseNumber=" + houseNumber + '\'';

        if (extraInfo != null) {
            result = result + ", extraInfo.length=" + extraInfo.length();
        } else {
            result = result + ", extraInfo=null";
        }

        return result + ", residents=" + Arrays.toString(residents) + '}';
    }
}

public class Practice {
    public static void main(String[] args) {
        Address house = new Address(); // создаём объект и инициализируем его поля
        house.city = "Stockholm";
        house.street = "Drottninggatan";
        house.houseNumber = 68;
        house.residents = new String[]{"Эмма Нильссон", "Ларс Эрикссон"};
        System.out.println(house.toString());
    }
}

```




Результат

```
Address{city='Stockholm', street='Drottninggatan', houseNumber=68',  
extraInfo=null, residents=[Эмма Нильссон, Ларс Эрикссон]}
```

Какие утверждения о методе **toString()** правильные?

- 1) Метод **toString()** возвращает хеш-код объекта.
- 2) Метод **toString()** возвращает текстовое представление объекта.
- 3) Чтобы получить как можно больше информации об объекте, в реализацию **toString()** нужно добавить вычисления.
- 4) Значения полей объекта нужно выводить через **toString()** строго полностью и без изменений.
- 5) При реализации **toString()** в разных классах следует придерживаться единого формата.

1) Метод `toString()` возвращает хеш-код объекта.

Реализация `toString()` по умолчанию использует хеш-код как часть возвращаемой строки, но этот метод не предназначен для вывода хеш-кода.

2) Метод `toString()` возвращает текстовое представление объекта.

Такое представление можно затем вывести в консоль или сохранить.

3) Чтобы получить как можно больше информации об объекте, в реализацию `toString()` нужно добавить вычисления.

Метод `toString()` должен возвращать только текущие значения полей. Вычислять их или менять не нужно.

4) Значения полей объекта нужно выводить через `toString()` строго полностью и без изменений.

Если поле содержит объёмные данные, то можно вывести только его длину. Какие-то данные, например, время или даты, можно дополнительно отформатировать.

5) При реализации `toString()` в разных классах следует придерживаться единого формата. Так легче будет читать отладочную информацию.



`println()` и `toString()`

Через метод `println()` можно сразу получить текстовое представление объекта. Вызывать `toString()` при этом необязательно. Так как все классы наследуют `toString()` от `Object`, метод `println()` может преобразовать в строку объект любого класса:

```
public class Address {
    public String city;
    public String street;
    public int houseNumber;

    public Address(String city, String street, int houseNumber) {
        this.city = city;
        this.street = street;
        this.houseNumber = houseNumber;
    }

    @Override
    public String toString() {
        return "Address{" +
            "city='" + city + '\'' +
            ", street='" + street + '\'' +
            ", houseNumber=" + houseNumber + '\'' + '}';
    }
}

public class Practice {
    public static void main(String[] args) {
        Address address = new Address("Астана", "Туркестан", 34);

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



Результат

```
Address{city='Астана', street='Туркестан', houseNumber=34 }
```

Если `toString()` не переопределён — то получим неинформативный результат, сгенерированный базовой реализацией:

```
public class Address {
    public String city;
    public String street;
    public int houseNumber;

    public Address(String city, String street, int houseNumber) {
        this.city = city;
        this.street = street;
        this.houseNumber = houseNumber;
    }
}

public class Practice {
    public static void main(String[] args) {
        Address address = new Address("Астана", "Туркестан", 34);

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



Результат

Address@33c7353a

Многие классы стандартной библиотеки **String**, **Integer**, **Double**, **Short** и другие уже имеют переопределённый **toString()**. При передаче их объектов в метод **println()** выводятся непосредственно значения, а не название класса с хеш-кодом. При конкатенации — оператор **+** также автоматически вызывает **toString()**.

```
public class Practice {  
    public static void main(String[] args) {  
        Integer number = 42;  
        Double secondNumber = 42.24;  
        String text = "Текст";  
        System.out.println(number);  
        System.out.println(secondNumber);  
        System.out.println(text);  
        System.out.println(text + number);  
    }  
}
```



Результат

42

42.24

Текст

Текст42

```
public class Note {
    public String label;
    public String text;

    @Override
    public String toString() {
        return "Note{label='" + label + "',text.length=" + text.length() + "}";
    }
}

public class Practice {
    public static void main(String[] args) {
        Note note1 = new Note();
        note1.label = "Купить продукты";
        note1.text = "помидоры, творог";
    }
}
```

Что будет выведено на консоль, если набрать команду:

```
System.out.println(note1.toString());
```

A) "Купить продукты: помидоры, творог"

B) Note@33c7353a

C) Note{label='Купить продукты',text.length=16}

A) "Купить продукты: помидоры, творог"

Реализация метода `toString()` в классе `Note` работает по-другому.

B) `Note@33c7353a`

Подобный текст возвращает базовая реализация `toString()`. Метод переопределён.

C) `Note{label='Купить продукты', text.length=16}`

Такой формат вывода `toString()` хорош тем, что показывает класс и значения конкретных полей.

```
public class Note {
    public String label;
    public String text;

    @Override
    public String toString() {
        return "Note{label='" + label + "',text.length=" + text.length() + "}";
    }
}

public class Practice {
    public static void main(String[] args) {
        Note note1 = new Note();
        note1.label = "Купить продукты";
        note1.text = "помидоры, творог";
    }
}
```

Что изменится, если метод `println` будет вызван так:

```
System.out.println(note1);
```

А) Будет ошибка.

В) Ничего не поменяется.

С) Будет выведена строка, состоящая из названия класса, символа '@' и хэш-кода объекта.

А) Будет ошибка.

Ошибки нет. Метод `println` умеет работать с объектами любых классов, а также значениями примитивных типов.

В) Ничего не поменяется.

Аргументом `println` может быть любой объект. Реализация метода `println` сама вызывает `toString()` у переданных объектов.

С) Будет выведена строка, состоящая из названия класса, символа '@' и хэш-кода объекта. Такие данные возвращаются, если метод `toString()` не переопределён.

```
public class Note {  
    public String label;  
    public String text;  
  
    @Override  
    public String toString() {  
        return "Note{label='" + label + "',text.length=" + text.length() + "}";  
    }  
}  
  
public class Practice {  
    public static void main(String[] args) {  
        Note note2 = new Note();  
        note2.label = "Адрес Асана - Бостандыкский пр. 10";  
        System.out.println(note2);  
    }  
}
```

Что будет выведено на консоль в данном случае?

A) Note{label='Адрес Асана - Бостандыкский пр. 10', text.length=0}

B) Ошибка

C) Note{label='Адрес Асана - Бостандыкский пр. 10'}

A) `Note{label='Адрес Асана - Бостандыкский пр. 10', text.length=0}`

Так не получится. В `text` будет `null`, а в реализации метода `toString()` нет обработки полей с пустой ссылкой.

В) Ошибка

В реализации `toString()` нет проверки поля `text` на `null`. Значит при вызове его метода `length()` - произойдет ошибка.

C) `Note{label='Адрес Асана - Бостандыкский пр. 10'}`

Реализация метода `toString()` предполагает вывод длины поля `text`, однако из-за того, что в нём нет значения - произойдет ошибка



Задача

https://github.com/practicetasks/java_tasks/blob/main/object_hashcode_equals/task_3/README.md



Решение

<https://gist.github.com/practicetasks/85126d7dcf25dffa391edbfdfdc08796>