Object

Класс: `java.lang.Object`

Описание

Базовый класс, от которого наследуют все остальные классы в языке Java. Любой объект Java, включая массивы, содержит все методы Object. Выражение любого типа (кроме числовых и boolean) можно присваивать переменной типа Object, а в качестве параметра метода, ожидающего Object, можно передавать вообще любое выражение, которое при необходимости завернётся в объект (например, `int` в `Integer`).

Строковое представление

У любого объекта есть метод `toString`:

```
String toString()
```

Именно этот метод используется при слиянии строк оператором +, если вместо строки передать произвольный объект. Исключение составляет значение `null`, которое при прибавлении к строке даст строку `"null"`. Большинство классов стандартной библиотеки, хранящих в себе данные **(кроме массивов!)**, переопределяют этот метод так, чтобы выводить удобное представление этих данных.

Кроме того, метод `toString` используется методами `print` и `println`, а также спецификатором `%s` в контексте `printf`. Поэтому в Java (в отличие от C и C++) `%s` в `printf` работает с любым выражением:

```
Object obj = "Hello";
LocalDate einsteinBirth = LocalDate.of(1879, Month.MARCH, 14);
List<Integer> numbers = Arrays.asList(1, 2);

System.out.println(obj + " World");
System.out.printf("Мои любимые числа - %s\n", numbers);
String einsteinFact = String.format("Эйнштейн родился %s", einsteinBirth);
System.out.println(einsteinFact);
```

Результат:

```
Hello World
Мои любимые числа - [1, 2]
Эйнштейн родился 1879-03-14
```

Реализация `toString()` по умолчанию, содержащаяся в классе `Object`, выводит полное имя класса и его хэш-код в шестнадцатиричном представлении. Например, следующая программа использует реализацию `toString` по умолчанию:

```
package com.hornsandhooves;

public class DefaultToString {
    public static void main(String[] args) {
        System.out.println(new DefaultToString());
    }
}
```

Результатом будет что-то вроде:

`com.hornsandhooves.DefaultToString@7d4991ad`

Важно! К сожалению, по историческим причинам массивы тоже используют реализацию по умолчанию:

```
int[] numberArray = { 1, 2 };
System.out.println("Мои любимые числа - " + numberArray);
// Мои любимые числа - [I@7d4991ad
```

Если же нужно распечатать именно содержимое массива, нужно использовать статический метод `Arrays.toString`:

```
int[] numberArray = { 1, 2 };
System.out.println("Мои любимые числа - " + Arrays.toString(numberArray));
// Мои любимые числа - [1, 2]
```

Имеет смысл переопределять `toString` в своих классах, чтобы было удобно отлаживать их внутреннее представление.

```
package com.hornsandhooves;

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

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

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

    public static void main(String[] args) {
        System.out.println(new Person("Остап", "Бендер"));
    }
}
```

Результат:

`Остап Бендер`

Отладчики в популярных IDE (например, Eclipse) используют `toString` как представление

объекта, например, в списке локальных переменных или отслеживаемых выражений.

Рефлексия

Рефлексия (reflection) — это подсистема платформы Java, позволяющая получать информацию о классах и объектах во время выполнения, а также обращаться к полям и методам классов и объектов, зная только их имена и типы параметров. Это очень мощный механизм, которого пока не имеет смысла касаться в полной мере. Отметим лишь, что основным классом, предоставляющим средства рефлексии, является класс `Class`, и у любого объекта можно запросить его реальный (а не объявленный) тип с помощью метода `getClass`:

`Class<? extends объявленный_тип_объекта> getClass()`

Например, можно написать вот так (мы помним, что `String` и `StringBuilder` являются подтипами `CharSequence`):

```
CharSequence str = "Hello";
CharSequence sb = new StringBuilder("World");
Class<? extends CharSequence> = str.getClass();  // String.class
Class<? extends CharSequence> = sb.getClass();   // StringBuilder.class
```

Здесь страшные письмена `<? extends CharSequence>` означают "тип `CharSequence` или некоторый его подтип". Таким образом, запись `Class<? extends CharSequence>` означает "объект `Class`, соответствующий типу `CharSequence` или какому-то его подтипу (например, `String` или `StringBuilder`), но какому именно, компилятор не знает".

У класса `Class` очень много методов. Вот лишь пара примеров:

`String getName()`

Возвращает полное имя класса вместе с именем пакета, через точку.

`String getSimpleName()`

Возвращает имя класса без пакета.

Особо любопытные могут использовать `getClass`, чтобы узнать реальное имя класса, экземпляром которого является объект, даже если мы работаем с ним через супертип (например, интерфейс `List`). Так может случиться, например, если фабричный метод прячет конкретный тип как деталь реализации — таков, например, метод `Arrays.asList`:

```
List<String> strings = new ArrayList<>();
List<Integer> numbers = Arrays.asList(1, 2);

strings.getClass().getName()  // java.util.ArrayList
numbers.getClass().getName()  // java.util.Arrays$ArrayList
```

Естественно, поскольку это деталь реализации, в будущих версиях JDK реальные имена классов, скрывающихся за фабричными методами, вполне могут измениться.

Каждому классу или интерфейсу в памяти всегда гарантированно соответствует ровно один экземпляр типа `Class`. Поэтому объекты типа `Class` можно сравнивать по `==`:

```
if ("Hello".getClass() == "World".getClass()) { ... }  // true
```

Сравнение и хэш-код

Метод `equals` сравнивает два объекта на равенство по значению:

`boolean equals(Object obj)`

Классы стандартной библиотеки, представляющие значения (например, `String` и `LocalDate`), переопределяют метод `equals`, и он, как правило, работает так, как мы ожидаем:

```
LocalDate nlLiberationDay = LocalDate.of(1945, Month.MAY, 5);
LocalDate ruVictoryDay = LocalDate.of(1945, Month.MAY, 9);

nlLiberationDay.plusDays(4) == ruVictoryDay       // false
nlLiberationDay.plusDays(4).equals(ruVictoryDay)  // true
```

Если же мы реализуем свой класс и хотим сравнивать его экземпляры по значению, то мы должны написать свой `equals`, потому что реализация по умолчанию (в `Object`) сравнивает объекты по ссылке! Вот её исходный код:

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

Реализация метода `equals` должна удовлетворять четырём требованиям, первые три из которых известны из математики как определение *отношения эквивалентности*:

1. **Рефлексивность**: Для всякого `x` должно выполняться условие `x.equals(x)`.
2. **Симметричность**: Для всяких `x` и `y` условие `x.equals(y)` должно выполняться тогда и только тогда, когда `y.equals(x)`.
3. **Транзитивность**: Для всяких `x` и `y` и `z`, если `x.equals(y)` и `y.equals(z)`, то `x.equals(z)`.
4. **"Ненулёвость"**: Для всякого `x != null` условие `x.equals(null)` должно быть ложным.

Важно! Нельзя вызвать метод `equals`, как и вообще никакой метод, у ссылки `null`:

```
String broken = null;
broken.equals("Hello")  // Выбросит NullPointerException
```

Если нужно сравнить между собой две ссылки, каждая из которых может быть равна `null`, можно использовать статический метод `Objects.equals` (не путать с `Object.equals`). Следующий вызов

`Objects.equals(x, y)`

возвращает `true` тогда и только тогда, когда:

- либо `x == null` и `y == null`,
- либо `x != null` и `x.equals(y)`.

Однако, прежде чем мы бросимся реализовывать `equals`, нужно прояснить один тонкий момент. Некоторым контейнерам (их называют *хэш-таблицами* для эффективности нужно не только сравнивать объекты на равенство, но и иметь возможность запросить у объекта числовое значение, которое всегда было бы равным для равных объектов, а для большинства разных объектов было бы разным. Это значение называется *хэш-кодом*, и его можно запросить у любого объекта:

`int hashCode()`

Детали реализации хэш-таблиц нам сейчас не важны. Важно усвоить одно правило: *при переопределении* `equals` *всегда нужно также переопределять* `hashCode`. Если переопределить `equals` и оставить стандартную реализацию `hashCode` (которая, как правило, возвращает разные значения для объектов с разными адресами), работа хэш-таблиц нарушится:

```
class Person {
    private final String firstName;
    private final String lastName;

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;  // оптимизация
        }

        if (!obj instanceof Person) {
            return false;
        }

        Person otherPerson = (Person) obj;
        return Objects.equals(firstName, otherPerson.firstName)
            && Objects.equals(lastName, otherPerson.lastName);
    }

    // Ошибка - не переопределили hashCode!
}
```

```
Set<Person> famousPeople = new HashSet<>();

Person newton = new Person("Исаак", "Ньютон");
Person newtonClone = new Person("Исаак", "Ньютон");
System.out.println(newton.equals(newtonClone));          // true

famousPeople.add(newton);
System.out.println(famousPeople.contains(newtonClone));  // false - ошибка!
```

Должно выполняться следующее условие:

> ***Если* объекты равны по `equals`, *то* у них должен быть одинаковый хэш-код.**

Обратное, вообще говоря, может быть неверным. Вот самая простая (и самая ужасная) реализация `hashCode`, которую только можно себе представить:

```
@Override
public int hashCode() {
    return 42;    // Никогда не делайте так!
}
```

Наш код заработает, но неэффективно. Букву контракта мы соблюли, но не дух. Действительно, у любых двух равных объектов будет одинаковый хэш-код. Проблема в том, что так вообще у любых двух объектов нашего класса будет одинаковый хэш-код. С точки

зрения производительности это совершенно неприемлемо. Желательно, чтобы как можно чаще было верно и обратное: если `x.equals(y) == false`, то `x.hashCode() != y.hashCode()`.

Подробные рекомендации по реализации `equals` и `hashCode` можно найти в книге Джошуа Блоха "Java. Эффективное программирование" (Effective Java). Здесь отметим лишь, что проще всего для реализации `hashCode` использовать статический метод `Objects.hash` *по тем же полям, что сравниваются в* `equals`. Например, для приведённого выше класса `Person` мы можем реализовать `hashCode` вот так:

```
@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}
```

Важно! Если вы реализуете интерфейс `Comparable` для упорядочения объектов, то, как правило, должно выполняться ещё одно правило: реализация метода `compareTo` должна быть совместимой с `equals`, то есть `x.equals(y)` должно быть верно тогда и только тогда, когда `x.compareTo(y) == 0`. Например, в нашем классе `Person` мы могли бы реализовать возможность сортировки сначала по фамилии, а затем по имени:

```
public class Person implements Comparable<Person> {
    ...

    @Override
    public int compareTo(Person other) {
       return Comparator.comparing(Person::lastName)
                        .thenComparing(Person::firstName);
    }

    @Override
    public String toString() { return firstName + " " + lastName; }

    ...
}
```

А вот пример использования такого порядка:

```
List<Person> quantumPhysicists = Arrays.asList(
    new Person("Эрвин", "Шрёдингер"),
    new Person("Нильс", "Бор"),
    new Person("Альберт", "Эйнштейн"),
    new Person("Поль", "Дирак"),
    new Person("Вернер", "Гейзенберг")
);

quantumPhysicists.sort(null);
quantumPhysicists.forEach(System.out::println);
```

Результат:

```
Нильс Бор
Вернер Гейзенберг
Поль Дирак
Эрвин Шрёдингер
Альберт Эйнштейн
```

У нашего класса `Person` всё хорошо с совместимостью `equals` и `compareTo`. А вот в стандартной библиотеке есть пример того, как делать не надо: метод `equals` класса `BigDecimal` обращает внимание на незначащие нули, а метод `compareTo` их игнорирует!

```
BigDecimal onePoint0 = new BigDecimal("1.0");
BigDecimal onePoint00 = new BigDecimal("1.00");

onePoint0.equals(onePoint00)     // false
onePoint0.compareTo(onePoint00)  // 0
```

Уничтожение объекта

Метод `Object.finalize` вызывается перед тем, как объект будет уничтожен сборщиком мусора. Он объявлен как `protected`, поэтому его нельзя вызвать из внешнего кода, можно только переопределить. По умолчанию он не делает ничего.

```
public class LastWords {
    @Override
    protected void finalize() {
        System.out.println("Мрачный жнец пришёл за мной");
    }

    public static void main(String[] args) {
        new LastWords();
        System.gc();      // Явно вызвали сборщик мусора
    }
}
```

Вообще говоря, нет гарантии того, что `finalize` будет вызван хоть когда-нибудь. Гипотетическая реализация JVM с бесконечной памятью имеет вполне законное право не только никогда не производить сборку мусора неявно, но и определить метод `System.gc` вот так:

```
public static void gc() { }
```

Поэтому, если класс держит какой-то ресурс, которые не контролирует сборщик мусора Java (а таково большинство ресурсов ОС — файлы, окна, сетевые соединения...), лучше освобождать эти ресурсы явно. Для этого принято реализовывать интерфейс `AutoCloseable` (или его подинтерфейс `Closeable`, появившийся раньше) и использовать объект внутри блока *try с ресурсами:*

```
public class HttpReader implements Closeable {
    private final Scanner scanner;

    public HttpReader(String host, int port) throws IOException {
        Socket socket = new Socket(host, port);
        String request = "GET / HTTP/1.1\r\n"
                + "Host: " + host + "\r\n"
                + "\r\n";

        socket.getOutputStream().write(
                request.getBytes(StandardCharsets.US_ASCII));
        socket.getOutputStream().flush();

        scanner = new Scanner(socket.getInputStream());
    }

    @Override
    public void close() throws IOException {
        scanner.close();
    }

    public static void main(String[] args) throws IOException {
        try (HttpReader reader = new HttpReader("google.com", 80)) {
            while (reader.scanner.hasNextLine()) {
                System.out.println(reader.scanner.nextLine());
            }
        }
    }
}
```

Пример вполне рабочий, хоть и абсолютно игрушечный. Сетевое соединение закрывается по выходу из блока `try` с помощью неявного вызова метода `close`.

```
HTTP/1.1 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Location: http://www.google.ru/?gfe_rd=cr&ei=QSvYVoNahMdguoKtyAw
Content-Length: 255
Date: Thu, 03 Mar 2016 12:17:05 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.ru/?gfe_rd=cr&amp;ei=QSvYVoNahMdguoKtyAw">here</A>.
</BODY></HTML>
```

Для перестраховки можно переопределить и метод `finalize`, чтобы закрыть связанные с объектом ресурсы даже в случае, если пользователь класса забыл вызвать `close`:

```
@Override
protected void finalize() {
    close();
}
```

Библиотечный класс `Socket`, кстати, так и делает, но в общем случае полагаться на это нельзя, потому что `finalize` будет вызван не сразу, а только позже, во время сборки мусора — если он вообще когда-нибудь будет вызван.