CODEGYM

Mentor-supported training program from Codegym

Java developer in 12 months

MODULE 2. JAVA CORE

Lesson #9
Object class

CODEGYM

Lesson plan

- Object class methods
- equals method
- hashCode method
- clone method
- finalize method
- getClass method
- What aremutable and immutable objects

CODEGYM

Object class methods

Object – base class for all classes. It doesn't hold much data but has several methods.

Object class methods are common to all classes. The Java developers took a few methods that they thought all classes should have and added them to the Object class.

Combined with polymorphism - the ability to override methods of the Object class in derived classes - this creates a very powerful tool.

Method	Description
`public String toString()`	Returns a string representation of an object.
`public native int hashCode()` `public boolean equals(Object obj)`	Two methods that are used to compare objects.
`public final native Class getClass()`	Returns a special object, which describes the current class.

`public final native void notify()`
`public final native void notifyAll()`
`public final native void wait(long timeout)`
`public final void wait(long timeout, int nanos)`
`public final void wait()`
	Methods for controlling access to an object from different threads. Thread synchronization control.

`protected void finalize()`	The method allows you to “release" native non-Java resources: close files, streams, etc.
`protected native Object clone()`	Method allows to clone an object: creates a duplicate of an object.

CODEGYM

toString() method

This method allows a string description of an object to be returned.

Its implementation in the Object class is very simple:

```
return getClass().getName() + "@" + Integer.toHexString(hashCode());
```

getClass() and hashCode() – are also methods of the Object class.

Here's a standard result of calling such a method.
`java.lang.Object@12F456`

The value of this method is that it can be overridden in any class and return a more necessary or a more detailed description of the object.

CODEGYM

Equals method

The purpose of the equals method is to determine if objects are identical internally, by comparing the internal contents of the objects.

The Object class has its own implementation of the equals method, which simply compares references:

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

But this method was also created so that developers would override it in their classes. After all, only the developer of the class knows what data is important, what to take into account when comparing, and what to ignore.

CODEGYM

hashCode method

The equals method has a big disadvantage – **it works too slowly.**

The hashCode method is used if a quick comparison is required.

The hashCode() method returns a specific number for each object. Which number - this is also decided by the developer of the class, as in the case of the equals method.

Next, you need to compare these hashCodes and only if the hashCodes are equal, compare the objects using equals.

The developer who implements the hashCode method needs to know the following things:

A) Two different objects can have the same hashCode.

B) Two equal objects (from the equals method's point of view) must have the same hashCode.

C) hashcodes must be chosen in such a way so that there are not many different objects with the same hashCode.

CODEGYM

Now to the most important

If you override the equals method, you must override the hashCode() method, subject to the three rules above.

The thing is that collections in Java, before comparing objects using equals, always look for / compare them using the hashCode() method.

And if identical objects have different hashcodes, then the objects will be considered different - comparison with equals will simply not happen.

CODEGYM

Introduction to wait, notify and notifyAll methods

Java has a built-in mechanism for managing access to shared resources (objects) from different threads.

A thread can declare some object as busy, and other threads will have to wait until the busy object is released. This is done using the synchronized keyword.

As soon as a thread enters a block marked synchronized, the monitor object is marked as busy, and other threads will have to wait for the monitor object to become free. The same monitor object can be used in different parts of the program.

It is customary to call a monitor an object that stores the busy/free state.

CODEGYM

The wait method

When our thread has locked the monitor, other threads are forced to wait, although their data may already be ready for work.

To solve this problem, the wait() method was invented. Calling this method causes the thread to release the monitor and then "pause".

The wait method can be called on a monitor object and only when that monitor is busy—i.e. inside the synchronized block. This temporarily stops the thread and frees up the monitor so that other threads can use it.

Method wait()	Explanation
`void wait(long timeout)`	The thread "freezes", but after the number of milliseconds passes, it automatically "unfreezes".
`void wait(long timeout, int nanos)`	The thread "freezes", but after the number of milliseconds and nanoseconds passes, it automatically "unfreezes".

CODEGYM

notify/notifyAll methods

notify/notifyAll methods can only be called on the monitor object and only when this monitor is busy – i.e. inside the synchronized block.

notifyAll method unfreezes all threads, which were paused using the given monitor object.

notify method “unfreezes” one random thread, and notifyAll – all «frozen» threads of the given monitor.

CODEGYM

clone method

The purpose of the clone method is to clone an object - i.e. create its clone/copy/duplicate.

If it is called, the Java machine will create and return a duplicate of the object on which this method was called.

Cloning of an object in the Object class is implemented very primitively - when cloning, only one new object is created: another object is simply created, and its fields are assigned the values of the fields of the sample object.

If the copied object contains references to other objects, then the references will be copied, no duplicates of those objects are created.

CODEGYM

Cloneable interface

This is the so-called marker interface, which does not contain any methods. It is used to mark (flag) some classes.

If the class developer believes that class objects can be cloned, he marks the class with this interface (inherits the class from Cloneable).

If the developer is not satisfied with the standard implementation of the clone method, he must write his own, which will duplicate the object in the correct way.

When calling the clone() method, Java checks to see if the object had the Cloneable interface. If so, it clones the object using the clone() method; if not, it throws a CloneNotSupportedException.

CODEGYM

finalize method

finalize() – a special method that is called on an object before the garbage collector destroys it.

The main purpose of this method is to release external non-Java resources used: close files, I/O streams, etc.

Unfortunately, this method does not justify the hopes laid upon it.

The Java machine can delay the destruction of the object, as well as the call to the finalize method, for as long as it likes. Moreover, it does not guarantee at all that this method will be called. In a bunch of situations, for the sake of "optimization“, it is not called.

Java 7 introduced a new construct to replace the finalize method. It's called try-with-resources. This is not exactly a replacement for finalize() - rather an alternative approach.

CODEGYM

getClass method

To interact with classes there is a special class and it is called - Class.

Every time the Java machine loads a new class into memory, it creates an object of type Class, through which you can get some information about the loaded class.

Each class and object has such a "class object" associated with it.

Example	Description
`Class clazz = Integer.class;`	Get “class object” from Integer class.
`Class clazz = int.class;`	Get “class object” from int class.
`Class clazz = "123".getClass();`	Get “class object” from an object of a String type.
`Class clazz = new Object().getClass();`	Get “class object” from and object of Object type.

Java code	Description
`Class s = int.class;` `String name = s.getName();`	Get class name.
`Class s = Class.forName("java.lang.String");`	Get class by its name.

`Object o1 = String.valueOf(1);`
`Object o2 = 123 + "T";`
`o1.getClass() == o2.getClass();`
	Compare classes of objects.

CODEGYM

Reflection

This is the ability of a class to obtain information about itself.
Java has special classes:
Field is a field, Method is a method, similar to Class for classes.

Since an object of the Class type makes it possible to get information about a class, then an object of type Field gives you information about a “class field”, and Method gives you information about a “class method”.

Java code	Description
`Class[] interfaces = List.class.getInterfaces();`	Get a list of “class objects” for interfaces of the List class
`Class parent = String.class.getSuperclass();`	Get a «class object» of a parent class for the String class
`Method[] methods = List.class.getMethods();`	Get a list of methods of the List class
`String s = String.class.newInstance();`	Create a new String class object

`String s = String.class.newInstance();`
`Method m = String.class.getMethod("length");`
`int length = (int) m.invoke(s);`
	Get the length method of the String class, and call it on the s string

CODEGYM

What are mutable and immutable objects

Objects that can be changed after creation are called **mutable**.
Objects that cannot be changed after they have been created are called **immutable**.

Immutable objects have many useful properties. But there are two that are typical for almost all immutable objects:

1) Immutable objects can be implemented much easier than mutable ones.
2) Immutable objects can be freely used simultaneously from different threads.

The String class is an immutable class. All objects of type String are immutable, which, however, does not prevent us from working with them. For example, the **toUpperCase**() method of the **String** class converts a string to uppercase (replaces all small letters with uppercase letters). **This method does not change the string itself, but returns a new string**, which is identical to the original one, only all characters are uppercase (large).

CODEGYM

Homework

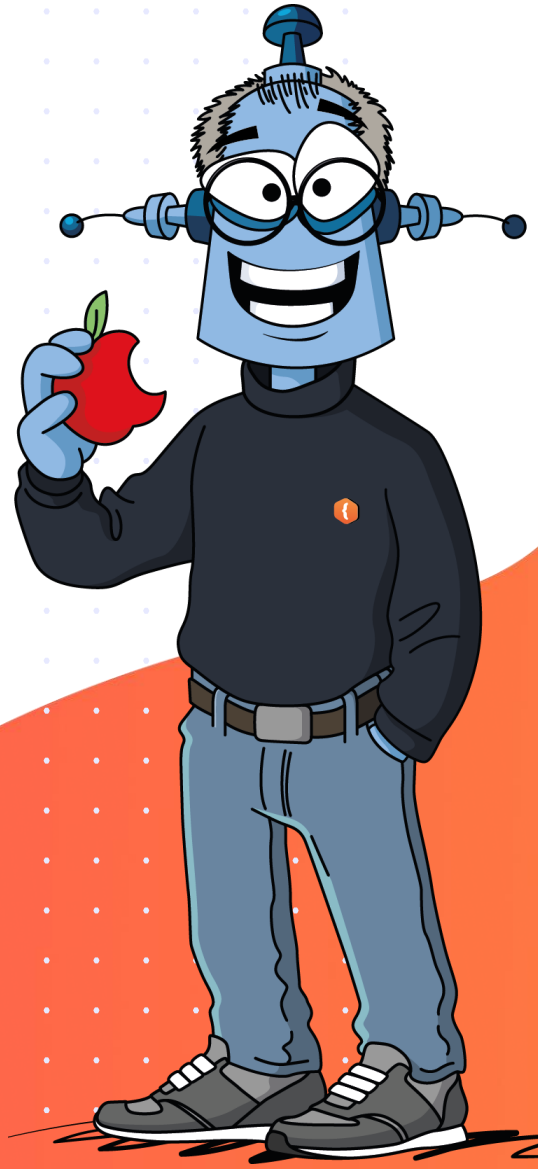
Module 2

Level 9 Object class

MODULE 2. JAVA CORE

CODEGYM

Questions and Answers



MODULE 2. JAVA CORE