CODEGYM

Mentor-supported training program from Codegym

Java developer in 12 months

MODULE 2. JAVA CORE

Lesson #13. Executor

CODEGYM

Lesson plan

- ThreadGroup
- Thread priorities
- Factory Method pattern
- Java Callable interface
- ExecutorService interface

CODEGYM

ThreadGroup

The concept of a “thread group” was introduced so that a thread could not stop or interrupt all other threads in a row.

A thread can only affect other threads that are in the same group.

ThreadGroup is the class that manages groups of threads. This approach allows you to protect the threads from unwanted changes.

CODEGYM

A few methods of the ThreadGroup class:

Method	Description
`String getName()`	**Returns the group name**
`ThreadGroup getParent()`	**Returns the parent's group name**
`void interrupt()`	**Terminates all threads in the group**
`boolean isDaemon()`	**Checks whether the group is a daemon**
`void setDaemon(boolean daemon)`	**Sets the group as a daemon**
`int activeCount()`	**Returns the number of live threads in the group and its sub-groups**
`int activeGroupCount()`	**Returns the number of live groups in the group and its sub-groups**
`int enumerate(Thread[] list)`	**Adds all live threads to an array and returns its count**
`int getMaxPriority()`	**Returns the maximum thread priority in the group**
`void setMaxPriority(int priority)`	**Allows to set maximum priority for threads in the group and its sub-groups**

CODEGYM

Thread priorities

In real jobs, the importance of different threads can vary greatly. Priority was invented to control the importance of different threads. Each thread has such a priority - this is a number from 1 to 10.

10 – maximum priority.
1 – lowest priority.

If a priority is not set, then a thread gets a priority of 5 (medium).

The priority of a thread does not greatly affect its operation, it is rather advisory in nature. If there are multiple sleeping threads to be started, then the Java machine will start the thread with the higher priority first.
The Java machine manages the threads as it sees fit. Low priority threads will not be idle. It's just that they will receive less time than others, but they will still be executed.

CODEGYM

The Thread class has two methods for working with priorities:

Method	Description
`void setPriority(int newPriority)`	Sets a new priority value
`int getPriority()`	Returns the current thread priority

The Thread class also has three constants:

```
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY = 5;
public static final int MAX_PRIORITY = 10;
```

CODEGYM

Factory Method pattern

A design template or pattern in software development is a repeatable architectural construct that represents a solution to a design problem within some recurring context.

Creational patterns are design patterns that deal with the process of creating objects. They make it possible to make the system independent of the method of creating, composing and presenting objects.

A factory method is a generative design pattern that defines a common interface for creating objects in a parent class, providing the ability to create these same objects to its heirs. During creation of an object descendants can determine which class to create.

CODEGYM

When to use a pattern?

1. When the types and dependencies of the objects that your code should work with are not known in advance.
2. When you want to save system resources by reusing already created objects instead of spawning new ones.
3. When you want to enable users to extend parts of your framework or library.

CODEGYM

Advantages and disadvantages of a pattern

Advantages:

1. Releases the class from being bound to specific transport classes.
2. Keeps the transport creation code in one place, making the code easier to maintain.
3. Simplifies the addition of new modes of transport to the program.
4. Implements the open/closed principle.

Disadvantages:

Can lead to large parallel class hierarchies, since each product class must have its own creator subclass.

CODEGYM

Callable

Java Callable(java.util.concurrent.Callable) interface represents an asynchronous task that can be executed by a separate thread.

The Callable interface is rather simple.
It only has a single call() method.

```
public interface Callable {
    V call() throws Exception;
}
```

CODEGYM

Future

Callable tasks return a java.util.concurrent.Future object. Using the Java Future object, we can find out the state of the Callable task and get the returned object.

It provides a get() method that can wait for the Callable to complete and then return the result.

Future provides a cancel() method to cancel the associated Callable task. There is a version of the get() method where we can specify a timeout for the result, so it's useful to avoid blocking the current thread for a longer time.

CODEGYM

Callable vs. Runnable

Callable is similar to Runnable in that both represent a task that is meant to be executed concurrently by a separate thread.

Callable differs from Runnable in that the run() method does not return a value and cannot throw exceptions (only RuntimeExceptions).

CODEGYM

ExecutorService interface

ExecutorService inherits from the Executor interface and adds functionality for managing the life cycle of threads.

It also includes the submit() method, which is similar to the execute() method, but more versatile. Overloaded versions of the submit() method can take either an executable (Runnable) or a callable (Callable) object.

The ExecutorService is very similar to a thread pool.
In fact, the ExecutorService implementation in the java.util.concurrent package is a thread pool implementation.

ExecutorService has the following implementations in the java.util.concurrent package:

ThreadPoolExecutor - Executes the specified tasks using one of its internal pool threads.

ScheduledThreadPoolExecutor is an ExecutorService that can schedule tasks to run after a delay or re-execution with a fixed time interval between each execution.

CODEGYM

Usage

There are several different ways to delegate tasks to an ExecutorService :
execute(Runnable);
submit(Runnable);
invokeAny();
invokeAll();

execute(Runnable) - takes a java.lang.Runnable object and executes it asynchronously.
submit(Runnable) - takes a Runnable implementation and returns a Future object.
Future object can be used to check if a Runnable has finished executing.
submit(Callable) - similar to submit(Runnable) but uses Callable instead of Runnable.
invokeAny() - takes a collection of callable objects. Calling this method returns the result of one of the called objects
invokeAll() - invokes all callables passed as parameters. It returns Future objects that can be used to get the execution results of each called object.

CODEGYM

Stopping ExecutorService

shutdown() method - when this method is called the ExecutorService will not immediately shut down, but it will no longer accept new tasks, and once all threads have finished executing the current tasks, the ExecutorService will exit. All tasks submitted to the ExecutorService prior to the call to shutdown() will be executed.

To close the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all running tasks at once and will skip any submitted but pending tasks. There are no guarantees regarding the completion of tasks. Perhaps they will stop, or perhaps they will be completed to the end.

The awaitTermination() method will block the thread that calls it until the ExecutorService has completely terminated or the specified timeout occurs. The awaitTermination() method is usually called after a call to shutdown() or shutdownNow().

CODEGYM

Homework

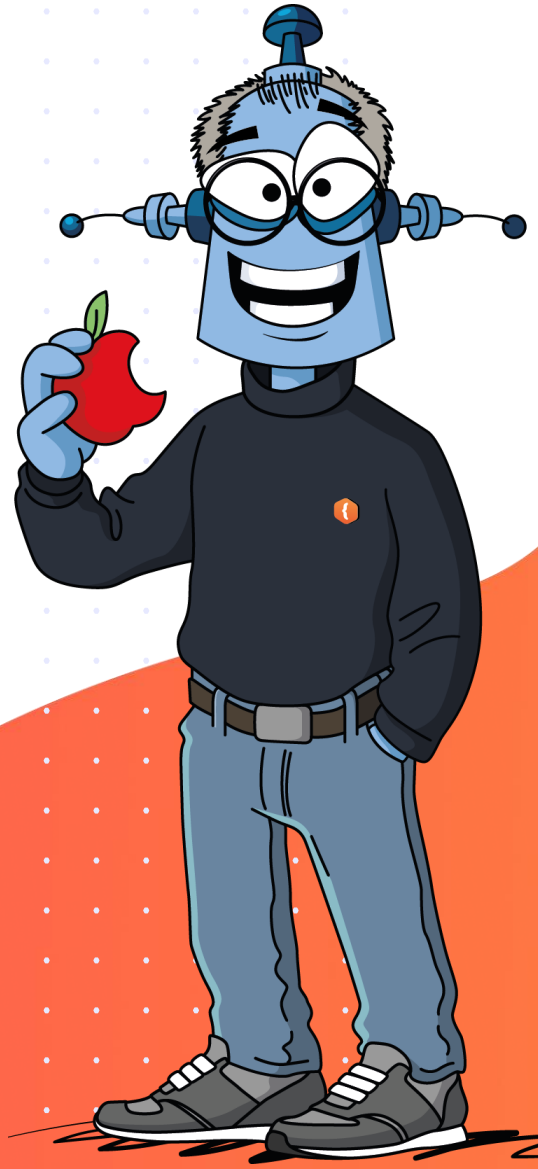
Module 2

Level 13. Executor

MODULE 2. JAVA CORE

CODEGYM

Questions and Answers



MODULE 2. JAVA CORE