CODEGYM

Mentor-supported training program from Codegym

Java developer in 12 months

MODULE 2. JAVA CORE

Lesson #12
Introduction to threads.
Part 2

CODEGYM

Lesson plan

- Disadvantages of multithreading
- Synchronized
- Volatile, yield, join, DeadLock
- Conditions for synchronizing memory

MODULE 2. JAVA CORE

CODEGYM

Disadvantages of multithreading

We shouldn't forget that there may be several threads, with each thread performing its own task.

Different threads change the state of different objects in accordance wit h the task that each
of them performs. Therefore, they can interfere with each other.

But the worst happens deep inside the Java machine. The apparent simultaneity of the threads is achieved due to the fact that the processor is constantly switching from one thread to another. And here comes the problem: switching can happen at the wrong moment.

CODEGYM

synchronized

Threads have been found to interfere with each other when they try to work together on shared objects and/or resources. Therefore, a special object was invented - a mutex.

It has two states - the object is free and the object is busy, or as they are also called - locked and unlocked.

The Java developers built this mutex into the Object class. You don't even have to create it. Every object has it.

CODEGYM

Code	Description

class MyClass{
  private String name1 = ”Mary";
  private String name2 = ”John";
 	
  public void swap(){
  synchronized (this){
  String s = name1;
  name1 = name2;
  name2 = s;
  }
  }
 	
}
	The swap method swaps the values of the name1 & name2 variables.
	
	What will happen if we call it from two threads simultaneously?

When one thread enters a block of code marked with the synchronized keyword, the Java machine immediately locks the mutex on the object that is specified in parentheses after the synchronized word.

No other thread will be able to enter this block until our thread leaves it. As soon as our thread exits the block marked as synchronized, the mutex automatically gets unlocked immediately and will be free to be captured by another thread.

We would like to draw your attention to the fact that both a piece of code and a method can be marked with the synchronized keyword.

We can combine all of this into a single class. The Thread class implements the Runnable interface, and it is sufficient to just override its run() method:

Another way of creating a thread	

`class Printer extends Thread{`
`private String name;`
`public Printer(String name){`
`  this.name = name;`
`  }`
`public void run(){`
`  System.out.println("I'm " + this.name);`
`  }`
`}`
	Extend from the Thread class, which implements the Runnable interface, and then override the run() method.

`public static void main(String[] args){`
`  Printer printer = new Printer("Mary");`
`  printer.start();`
 	
`Printer printer2 = new Printer("John");`
`printer2.start();`
`}`
	Create two threads, each of which is based on an object of the **Printer** type.

CODEGYM

volatile

The processor reads data from memory, modifies it, and writes it back to memory. To speed up the processor, its own "fast" memory was built - the cache.

To speed up its work, the processor copies the most frequently used variables from the memory area to its cache and makes all their changes in this fast memory. And then - copies back to the "slow" memory. Slow memory keeps the old(!) (unchanged) values of variables all this time.

And then there might be a problem. One thread changes the variable, and the second thread "does not see" this change, because it was done in quick memory. This is a consequence of the threads not having access to each other's cache. (A processor often contains several independent cores, and threads can physically run on different cores.)

The special **volatile** keyword was invented. Putting this keyword before a variable definition forced it to always read and write that variable only to the regular (slow) memory.

CODEGYM

yield

yield – static method of the Thread class

The processor is constantly switching between threads. Each thread is allocated a small piece of processor time, called a quantum.

When this time expires, the processor switches to another thread and starts executing its commands. Calling the Thread.yield() method allows you to prematurely end the current thread's time slice, or, in other words, switches the processor to the next thread.

CODEGYM

join

One thread can call the join method on the second thread's object. As a result, the first thread (which called the method) suspends its work until the end of the second thread (on the object of which the method was called).

It is worth distinguishing between two things here: we have a thread - a separate process for executing commands, and there is an object of this thread (a Thread object).

CODEGYM

DeadLock

DeadLock — mutual deadlock occurs when:

A. Each thread needs to acquire both mutexes during its operation.
B. The first thread has captured the first mutex and is waiting for the second to become free.
C. The second thread has captured the second mutex and is waiting for the first one to become free.

CODEGYM

Homework

Module 2

Level 12

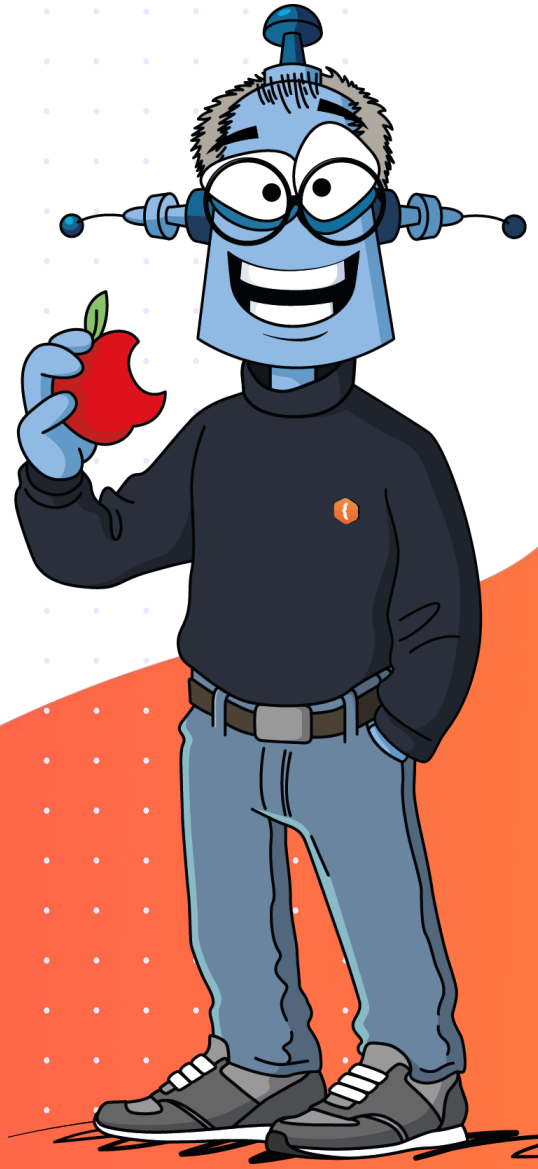
Introduction to threads.

Part 2

MODULE 2. JAVA CORE

CODEGYM

Questions and Answers



MODULE 2. JAVA CORE