**Programare Avansata pe Obiecte  
Laborator 12**

**Pahontu Bogdan-Ionut**

**E-mail:** [**pahontubogdan@gmail.com**](mailto:pahontubogdan@gmail.com)

Contents

[1. Threads 3](#_Toc9294970)

[1.1. Thread concurrency 3](#_Toc9294971)

[1.2. Creating a Thread 4](#_Toc9294972)

[1.3. Thread states 4](#_Toc9294973)

[1.4. Thread methods 5](#_Toc9294974)

[1.5. Thread class vs Runnable Interface 5](#_Toc9294975)

[1.6. Creating a Thread with Executor Service 5](#_Toc9294976)

[1.7. Shutting down a thread 6](#_Toc9294977)

[1.8. Synchronizing methods 7](#_Toc9294978)

[2. Sockets 7](#_Toc9294979)

[2.1. Socket class 7](#_Toc9294980)

[2.2. ServerSocket class 8](#_Toc9294981)

[2.3. Simple example 8](#_Toc9294982)

[3. Tasks 11](#_Toc9294983)

# Threads

* A thread is the smallest unit of execution that can be scheduled by the operating system
* A process is a group of associated threads that execute in the same, shared environment
* A system thread is created by the JVM and runs in the background of the application (ex: garbage collection)
* A user-defined thread is one created by the application developer to accomplish a specific task

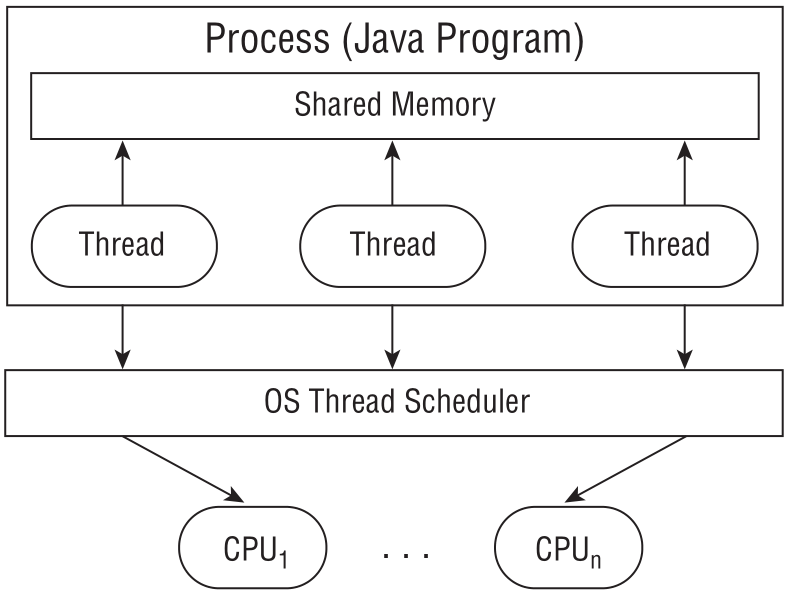


Figure 1 – Threads Digram

## Thread concurrency

* The property of executing multiple threads and processes at the same time is referred to as concurrency;
* Operating systems use a thread scheduler to determine which threads should be currently executing;
* A context switch is the process of storing a thread’s current state and later restoring the state of the thread to continue execution -> allows multiple threads to access the CPU;
* A thread priority is a numeric value associated with a thread that is taken into consideration by the thread scheduler when determining which threads should currently be executing;

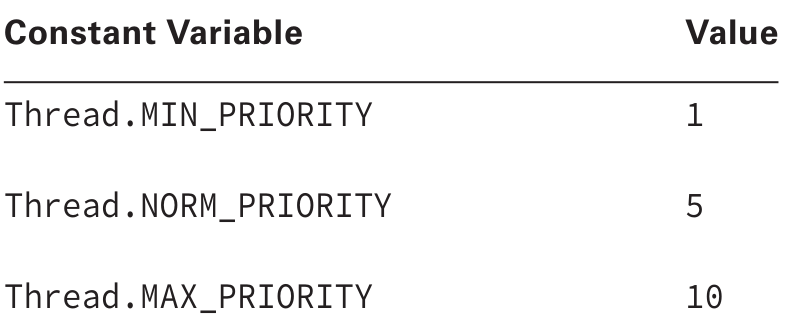


Figure 2 – Threads Priority

## Creating a Thread

* There are **2 + 1 ways** to define threads/workers in Java:
  1. Extend the java.lang.Thread class (run() method is void) -> Create a class that extends Thread and overrides the run() method.
  2. Implement the java.lang.Runnable interface or use its lambda expression (run method is void) -> Provide a Runnable object or lambda expression to the Thread constructor. You can also pass it to an ExecutorService and then call its execute() or submit() methods.
  3. Implement the java.lang.Callable interface or use its lambda expression (call() method is **NOT** void) -> similar to Runnable but you can pass and receive parameters to/from the call() method and you submit the Callable class to a java.util.concurrent.ExecutorService and NOT a Thread. From the ExecutorService submit() method you get a asynchronous promise, a Future object which will eventually provide you with a result when you call the future.get() method.
* For 1 and 2 remember to start the task with the Thread.start() method and not run()

## Thread states

* **New** - When we create an instance of Thread class, a thread is in a new state.
* **Running** - The Java thread is in running state.
* **Suspended** - A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
* **Blocked** - A Java thread can be blocked when waiting for a resource.
* **Terminated** - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.

## Thread methods

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| getName | Obtain thread’s name |
| getPriority | Obtain thread’s priority |
| isAlive | Determine if a thread is still running |
| join | Wait for a thread to terminate |
| run | Entry point for the thread |
| sleep | Suspend a thread for a period of time |
| start | Start a thread by calling its run method |

## Thread class vs Runnable Interface

* If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
* We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

## Creating a Thread with Executor Service

* The Concurrency API, introduced the ExecutorService, which creates and manages threads
* Obtain an instance of an ExecutorService interface -> you send the service tasks to be processed

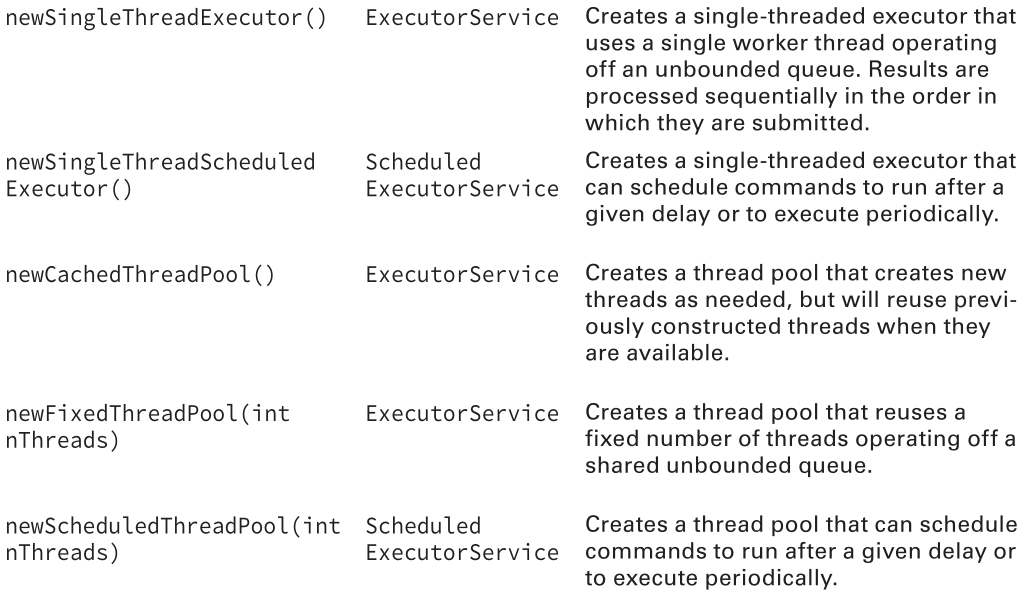


Figure 3 – Creating a Thread

## Shutting down a thread

* Once you have finished using a thread executor, it is important that you call the **shutdown**() method.
* A thread executor creates a non-daemon thread on the first task that is executed, so failing to call **shutdown**() will result in your application never terminating
* **shutdown**() does not actually stop any tasks that have already been submitted to the thread executor
* **shutdownNow**() attempts to stop all running tasks and discards any that have not been started yet -> returns a List<Runnable> of tasks that were submitted to the thread executor but that were never started
* Good practice to close a thread executor in a finally block

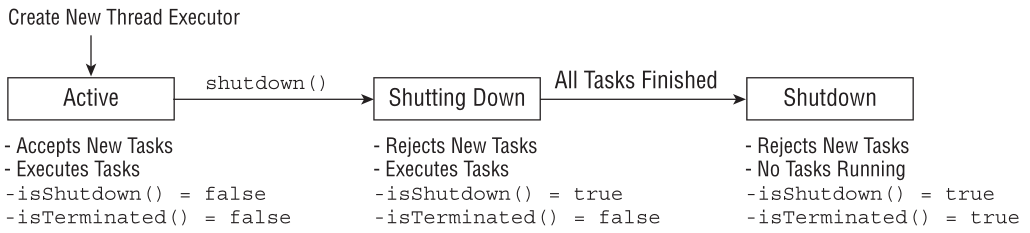


Figure 4 – Shutting down a thread

## Synchronizing methods

We can add the synchronized modifier to any instance method to synchronize automatically on the object itself. For example, the following two method definitions are equivalent:

**private void incrementAndReport() {**

**synchronized(this) {**

**System.out.print((++sheepCount)+" ");**

**}**

**}**

**private synchronized void incrementAndReport() {**

**System.out.print((++sheepCount)+" ");**

**}**

# Sockets

A socket is an abstraction through which an application may send and receive data, in much the same way as an open ﬁle handle allows an application to read and write data to stable storage. A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network. Information written to the socket by an application on one machine can be read by an application on a different machine and vice versa.

The **client** in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

## Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public InputStream getInputStream() | returns the InputStream attached with this socket. |
| 2) public OutputStream getOutputStream() | returns the OutputStream attached with this socket. |
| 3) public synchronized void close() | closes this socket |

## ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public Socket accept() | returns the socket and establish a connection between server and client. |
| 2) public synchronized void close() | closes the server socket. |

## Simple example

In the following code will be presented a simple example with a server socket and a client socket. The server will display the messages sent by client.

Server code:

**public class** SocketServerEx  
{  
 *//initialize socket and input stream* **private** Socket **socket** = **null**;  
 **private** ServerSocket **server** = **null**;  
 **private** DataInputStream **in** = **null**;  
  
 *// constructor with port* **public** SocketServerEx(**int** port)  
 {  
 *// starts server and waits for a connection* **try** {  
 **server** = **new** ServerSocket(port);  
 System.***out***.println(**"Server started"**);  
  
 System.***out***.println(**"Waiting for a client ..."**);  
  
 **socket** = **server**.accept();  
 System.***out***.println(**"Client accepted"**);  
  
 *// takes input from the client socket* **in** = **new** DataInputStream(  
 **new** BufferedInputStream(**socket**.getInputStream()));  
  
 String line = **""**;  
  
 *// reads message from client until "Over" is sent* **while** (!line.equals(**"Over"**))  
 {  
 **try** {  
 line = **in**.readUTF();  
 System.***out***.println(line);  
  
  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
 System.***out***.println(**"Closing connection"**);  
  
 *// close connection* **socket**.close();  
 **in**.close();  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
  
 **public static void** main(String args[])  
 {  
 SocketServerEx server = **new** SocketServerEx(5000);  
 }  
}

Client code:

**public class** ClientSocketEx  
{  
 *// initialize socket and input output streams* **private** Socket **socket** = **null**;  
 **private** DataInputStream **input** = **null**;  
 **private** DataOutputStream **out** = **null**;  
  
 *// constructor to put ip address and port* **public** ClientSocketEx(String address, **int** port)  
 {  
 *// establish a connection* **try** {  
 **socket** = **new** Socket(address, port);  
 System.***out***.println(**"Connected"**);

*// takes input from terminal* **input** = **new** DataInputStream(System.***in***);  
  
 *// sends output to the socket* **out** = **new** DataOutputStream(**socket**.getOutputStream());  
 }  
 **catch**(UnknownHostException u)  
 {  
 System.***out***.println(u);  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
  
 *// string to read message from input* String line = **""**;  
  
 *// keep reading until "Over" is input* **while** (!line.equals(**"Over"**))  
 {  
 **try** {  
 line = **input**.readLine();  
 **out**.writeUTF(line);  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
  
 *// close the connection* **try** {  
 **input**.close();  
 **out**.close();  
 **socket**.close();  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
  
 **public static void** main(String args[])  
 {  
 ClientSocketEx client = **new** ClientSocketEx(**"127.0.0.1"**, 5000);  
 }  
}

# Tasks

**Task 1:**

* Implement a socket server that will allow connection of multiple clients and prints the messages received from these clients;
* Implement a socket client that will be able to connect to a socket server and send a message.
* Start the socket server.
* Create two clients that will connect to the server and send some messages;

Hint: use Threads

**Task 2:**

* Change the previous example and add “Read-write both sides”.
* Case: The client will connect to the server and send a message. The server will listen to the client and response to the client.