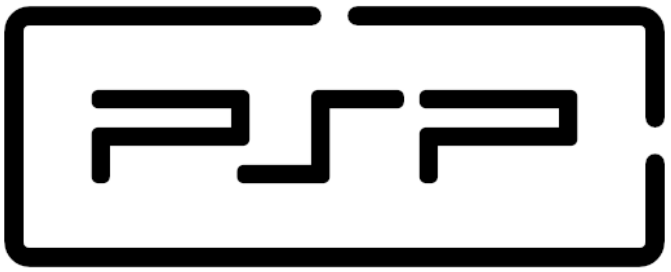

Process and Service Programming

3.1. Java classes for threads

I.E.S.
Doctor Balmis

3.1. Java classes for threads

2.1.1. Runnable Interface

A Java Thread can execute your Java code inside your Java application.

When a Java application is started its main() method is executed by the main thread - a special thread that is created by the Java VM to run your application. From inside your application you can create and start more threads which can execute parts of your application code in parallel with the main thread.

Java threads are objects like any other Java objects. Threads are instances of class java.lang.Thread, or instances of subclasses of this class. In addition to being objects, java threads can also execute code.

The first way to specify what code a thread should run is by creating a class that implements the `java.lang.Runnable` interface.

The Runnable interface is a standard Java Interface that comes with the Java platform. The Runnable interface only has a single method run().

> java.lang.Runnable specification

Whatever the thread is supposed to do when it executes must be included in the implementation of the run() method. There are three ways to implement the Runnable interface:

- Create a Java class that implements the Runnable interface.
- Create an anonymous class that implements the Runnable interface.
- Create a Java Lambda that implements the Runnable interface.

All three options are explained in the following sections.

Java Class Implements Runnable

The first way to implement the Java Runnable interface is by creating your own Java class that implements the Runnable interface. Here is an example of a custom Java class that implements the Runnable interface:

```java
public class MyRunnable implements Runnable {

  public void run(){
      System.out.println("MyRunnable running");
  }
}
```

All this Runnable implementation does is to print out the text MyRunnable running. After printing that text, the run() method exits, and the thread running the run() method will stop.

Anonymous Implementation of Runnable

You can also create an anonymous implementation of Runnable. Here is an example of an anonymous Java class that implements the Runnable interface:

```java
Runnable myRunnable =
    new Runnable(){
        public void run(){
            System.out.println("Runnable running");
        }
    }
```

Apart from being an anonymous class, this example is quite similar to the example that used a custom class to implement the Runnable interface.

Java Lambda Implementation of Runnable

The third way to implement the Runnable interface is by creating a Java Lambda implementation of the Runnable interface. This is possible because the Runnable interface only has a single unimplemented method, and is therefore practically (although possibly unintentionally) a functional Java interface.

Here is an example of a Java lambda expression that implements the Runnable interface:

```java
Runnable runnable =
        () -> { System.out.println("Lambda Runnable running"); };
```

Calling the run method on a Runnable class

Look at this sample code of Runnable implementation

```java
public class LiftOff implements Runnable {
    private int countDown = 10;
    private static int taskCount = 0;
    private final int id = taskCount;

    public LiftOff() {}

    public LiftOff(int countDown) {
        this.countDown = countDown;
    }

    @Override
    public void run() {
        while (countDown > 0) {
            System.out.println("#" + id + " (" + countDown + ")" );
            countDown--;
        }
        System.out.println("LiftOff (" + id + ")");
    }

    public static void main(String[] args) {
        LiftOff launch = new LiftOff();
        launch.run();
        System.out.println("Waiting for LiftOff!");
    }
}
```

What's wrong with previous execution

Is the "Waiting for LiftOff!" placed in the right place?

Try to create more instances of LiftOff and run them all?

Is the application doing something different to a single threaded application? What can you notice from the program output?

2.1.2 Thread subclass

The second way to specify what code a thread is to run, is to create a subclass of `java.lang.Thread` and override the run() method. The run() method is what is executed by the thread after you call

`start()` .

> java.lang.Thread specification

Here is an example of creating a Java Thread subclass:

```java
public class MyThread extends Thread {
  public void run(){
      System.out.println("MyThread running");
  }
}
```

To create and start the above thread you can do like this:

```java
MyThread myThread = new MyThread();
myTread.start();
```

The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU. When the run() method executes it will print out the text "MyThread running".

You can also create an anonymous subclass of Thread like this:

```java
Thread thread = new Thread(){
  public void run(){
    System.out.println("Thread Running");
  }
}

thread.start();
```

This example will print out the text "Thread running" once the run() method is executed by the new thread.

> **LiftOff example**
>
> Copy the original LiftOff example and now make it extends Thread class.
>
> Is the "Waiting for LiftOff!" placed in the right place? Is it working as it's supposed to?
>
> Try to create more instances of LiftOff and run them all is the application doing something different to a single threaded application? What can you notice from the program output?

2.1.3 Starting a Thread With a Runnable

To have the run() method executed by a thread, pass an instance of a class, anonymous class or lambda expression that implements the Runnable interface to a Thread in its constructor. Here is how that is done:

```java
Runnable runnable = new MyRunnable(); // or an anonymous class, or lambda...

Thread thread = new Thread(runnable);
thread.start();
```

When the thread is started it will call the run() method of the MyRunnable instance (see previous examples) instead of executing it's own run() method. The above example would print out the text "MyRunnable running".

> **Information**
>
> Hence, there are two ways to specify what code the thread should execute.
>
> - The first is to create a subclass of Thread and override the run() method.
> - The second method is to pass an object that implements Runnable to the Thread constructor.

```java
public class EjemploThread extends Thread {
  public void run() {
    // Código del hilo
  }

  public static void main(String[] args) {
    EjemploThread hilo = new EjemploThread();
    hilo.start();
  }
}
```

```java
public class EjemploRunnable implements Runnable {
  public void run() {
    // Código del hilo
  }

  public static void main(String[] args) {
    Thread hilo = new Thread(new EjemploRunnable());
    hilo.start();
  }
}
```

Subclass or Runnable?

There are no rules about which of the two methods is the best. Both methods works. **The preferred method is implementing Runnable**, and handing an instance of the implementation to a Thread instance.

A few reasons against extending Thread

- When extending the Thread class, we're not overriding any of its methods. Instead, we override the method of Runnable (which Thread happens to implement). This is a clear violation of IS-A Thread principle.
- Creating an implementation of Runnable and passing it to the Thread class utilizes composition and not inheritance – which is more flexible
- After extending the Thread class, we can't extend any other class From Java 8 onwards, Runnables can be represented as lambda expressions

Common Pitfall: Calling run() Instead of start()

When creating and starting a thread a common mistake is to call the run() method of the Thread instead of start(), like this:

```
Thread newThread = new Thread(MyRunnable()); newThread.run(); //should be start();
```

or

```
MyRunnable runnable = new MyRunnable(); runnable.run();
```

At first you may not notice anything because the Runnable's run() method is executed like you expected. However, it is **NOT executed by the new thread** you just created. Instead the run() method is executed by the thread that created the thread. In other words, the thread that executed the above two lines of code. To have the run() method of the MyRunnable instance called by the new created thread, newThread, **you MUST call the newThread.start() method**.

2.1.4 Thread class methods

If we take a look at the Thread class definition, we will find many methods. We must be careful because some of those methods like stop(), suspend(), resume() and destroy() are `deprecated`.

Next we can see the most commonly used methods of Thread class:

Method	Description
start()	Makes a thread execute the code in the run method()
boolean isAlive()	Checks if the thread is alive or not
sleep(long ms)	Changes the thread state to blocked for the ms specified

Method	Description
run()	Is the thread code to be run. It is called by the start method. It represents the lifecycle of a thread.
String toString()	Returns a readable representation a thread [threadName, priority, threadGroupName]
long getId()	Returns the thread id
void yield()	Makes the thread stop running at the moment going back to the queue and allowing other threads to be executed.
void join()	Called from another thread, waits for this thread to die
String getName()	Gets the thread name
String setName(String name)	Sets a name for the thread
int getPriority()	Gets the thread priority
setPriority(int p)	Sets the thread priority
void interrupt()	Interrupts the thread executions causing a InterruptedException
boolean interrupted()	Checks if a thread has been interrupted
Thread.currentThread()	STATIC method returns a reference to the thread that is running this code
boolean isDaemon()	Checks if thread is a daemon. A low-level process running independently from its process. A process can finish while a daemon thread is still running
setDaemon(boolean on)	Makes a thread turn into a daemon. By default all threads are user-threads when they are created.
int activeCount()	Returns the number of active threads in the thread group where the thread belongs to.
Thread.State getState()	Returns the thread state, one of NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING or TERMINATED.

Thread also has up to 9 constructors, most of them getting a Runnable object as parameter along with the thread name and the tread group.

Thread constructors
Thread()
Thread(Runnable target)
Thread(String name)

Thread constructors
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, Runnable target, String name, long stackSize, boolean inheritThreadLocals)

Here we can see an example of some of these methods in use

```java
public class ThreadMethodsExample extends Thread {

    ThreadMethodsExample (ThreadGroup group, String name) {
        // Call to parent class constructor with group and thread name
        super(group, name);
    }

    @Override
    public void run() {
        String threadName = Thread.currentThread().getName();
        System.out.println("["+threadName+"] " + "Inside the thread");
        System.out.println("["+threadName+"] " + "Priority: "
          + Thread.currentThread().getPriority());
        Thread.yield();
        System.out.println("["+threadName+"] " + "Id: "
          + Thread.currentThread().getId());
        System.out.println("["+threadName+"] " + "ThreadGroup: "
          + Thread.currentThread().getThreadGroup().getName());
        System.out.println("["+threadName+"] " + "ThreadGroup count: "
          + Thread.currentThread().getThreadGroup().activeCount());
    }

    public static void main(String[] args) {
        // main thread
        Thread.currentThread().setName("Main");
        System.out.println(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().toString());

        ThreadGroup even = new ThreadGroup("Even threads");
        ThreadGroup odd = new ThreadGroup("Odd threads");

        Thread localThread = null;
        for (int i=0; i<10; i++) {
            localThread = new ThreadMethodsExample((i%2==0)?even:odd, "Thread"+i);
            localThread.setPriority(i+1);
```

```
            localThread.start();
        }

        try {
            localThread.join(); // --> Will wait until last thread ends
                                // like a waitFor() for processes
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            System.err.println("The main thread was interrupted while waiting for "
              + localThread.toString() + "to finish");
        }
        System.out.println("Main thread ending");
    }
}
```

As you can see in the code above, the static method Thread.currentThread() should be called in order to get the instance of the current thread running each statement, as there are many threads running the same code at the same time.

In the previous example we have used just one class for the new threads and for the main thread. And that's not the usual way to run threads. It's a better practice to split the code in separate classes.

Also note that the Thread (or Runnable) class can have it's own constructor to set its local properties or call the superclass constructors.

> **Split the code in two classes**
>
> Copy the code from the ThreadMethodsExample and split in in two classes. One containing the thread class and the other just having the main method and the calls to create and launch the processes.
>
> Next, change the ThreadMethodsExample to implement the Runnable interface and make the appropriate changes in the other class to make it work again.

Pay attention to that even if the threads are started in sequence (1, 2, 3 etc.) they may not execute sequentially, meaning thread 1 may not be the first thread to write its name to System.out. This is because the threads are in principle executing in parallel and not sequentially. The JVM and/or operating system determines the order in which the threads are executed. This order does not have to be the same order in which they were started nor each time the are run.

Pause a thread

A thread can pause itself by calling the static method **Thread.sleep()**. The sleep() takes a number of milliseconds as parameter. The sleep() method will attempt to sleep that number of milliseconds before resuming execution. The Thread sleep() is not 100% precise, but it is pretty good still. Here is an

example of pausing a Java thread for 3 seconds (3.000 milliseconds) by calling the Thread sleep() method:

```java
try {
    Thread.sleep(3000L);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

> **Real systems simulation**
>
> This method is going to be used in activities to simulate time lapses and speed up the real systems simulation. For instance, we can set that each *real* hour is just a second in the simulation, so one day will be reduced to 24 seconds.
>
> Also it's interesting when we need to set random time lapses for each thread, in order to get a realistic simulation of events in the real system.

Random numbers within a specific range of type integer, float, double, long, and boolean can be generated in Java.

There are three methods to generate random numbers in Java.

Method 1: Using random class

We can use the `java.util.Random` class to generate random numbers, following the steps below:

- Import the class java.util.Random
- Make the instance of the class Random, i.e., Random rand = new Random()
- Invoke one of the following methods of rand object:
  - nextInt(upperbound) generates random numbers in the range 0 to upperbound-1.
  - nextFloat() generates a float between 0.0 and 1.0.
  - nextDouble() generates a double between 0.0 and 1.0.

if we use the netxInt invocation with the bound parameter, we'll get numbers within a range

> int randomWintNextIntWithinARange = random.nextInt(max)

This will give us a number between *0 (inclusive)* and *max* (exclusive). The bound parameter must be greater than 0. Otherwise, we'll get a java.lang.IllegalArgumentException.

Method 2: Using Math.random

For generating random numbers within a range using Math.random(), follow the steps below:

- Declare the minimum value of the range
- Declare the maximum value of the range
- Use the formula Math.random()*(max-min)+min to generate values with the min and the max value inclusive.

The value returned by Math.random() is in the range 0 to 1 inclusive.

To generate a random value between 0 and an upper limit (50)

```
Math.random()*50
```

To generate a random value between 1 and an upper limit (50)

```
Math.random()*49+1
```

To generate a random bounded value, let's say between 200 and 500

```
Math.random()*300+200
```

Method 3: Use ThreadLocalRandom

The `java.util.Random` class doesn't perform well in a multi-threaded environment.

In a simplified way, the reason for the poor performance of Random in a multi-threaded environment is due to contention – given that multiple threads share the same Random instance.

To address that limitation, Java introduced the `java.util.concurrent.ThreadLocalRandom` for generating random numbers in a multi-threaded environment.

We just need to call `ThreadLocalRandom.current()` method, and it will return the instance of ThreadLocalRandom for the current thread. We can then generate random values by invoking available instance methods of the class.

To generate a random int value without any bounds:

```
int unboundedRandomValue = ThreadLocalRandom.current().nextInt());
```

To generate a random bounded int value, meaning a value between a given lower and upper limit.

```
int boundedRandomValue = ThreadLocalRandom.current().nextInt(0, 100);
```

Please note, 0 is the inclusive lower limit and 100 is the exclusive upper limit.

We can generate random values for long and double by invoking `nextLong()` and `nextDouble()` methods in a similar way as shown in the examples above.

Threads priority management

In Java, a thread's priority is an integer in the range 1 to 10. *The larger the integer, the higher the priority*. The thread scheduler uses this integer from each thread to determine which one should be allowed to execute. The Thread class defines three types of priorities:

- Minimum priority
- Normal priority
- Maximum priority

The Thread class defines these priority types as constants `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`, with values 1, 5, and 10, respectively. **NORM_PRIORITY is the default priority for a new Thread**.

Java's Thread class provides methods for checking the thread's priority and for modifying it.

The `getPriority()` instance method returns the integer that represents its priority.

The `setPriority()` instance method takes an integer between 1 and 10 for changing the thread's priority. If we pass a value outside the 1-10 range, the method will throw an error.

When we create a Thread, it inherits its default priority. When multiple threads are ready to execute, the JVM selects and executes the Runnable thread that has the highest priority. If this thread stops or becomes not runnable, the lower-priority threads will execute. In case two threads have the same priority, the JVM will execute them in FIFO order.

There are two scenarios that can cause a different thread to run:

- A thread with higher priority than the current thread becomes runnable
- The current thread exits the runnable state or yields (temporarily pause and allow other threads)

In general, at any time, the highest priority thread is running. But sometimes, the thread scheduler might choose low-priority threads for execution to avoid starvation.

```java
class U3S3_HiloPrioridad1 extends Thread {
  private int c = 0;
  private boolean stopHilo = false;
  public int getContador () {
    return c;
  }
  public void pararHilo() {
    stopHilo = true;
  }
  public void run() {
    while (!stopHilo) c++;
  }
}

public class U3S3_EjemploHiloPrioridad1 {
  public static void main(String args[]) {
    U3S3_HiloPrioridad1 h1 = new U3S3_HiloPrioridad1();
```

```
18      U3S3_HiloPrioridad1 h2 = new U3S3_HiloPrioridad1();
19      U3S3_HiloPrioridad1 h3 = new U3S3_HiloPrioridad1();
20
21      h1.setPriority(Thread.NOR_PRIORITY);
22      h2.setPriority(Thread.MAX_PRIORITY);
23      h3.setPriority(Thread.MIN_PRIORITY);
24
25      h1.start();
26      h2.start();
27      h3.start();
28
29      try {
30        Thread.sleep(10000);
31      } catch (exception e) {}
32
33      h1.pararHilo();
34      h2.pararHilo();
35      h3.pararHilo();
36
37      System.out.println("h2 (Prio. Máx: "+h2.getContador());
38      System.out.println("h1 (Prio. Nomral: "+h1.getContador());
39      System.out.println("h3 (Prio. Mínima: "+h3.getContador());
40    }
41  }
```