Threads programming

A **thread** is the **smallest processing unit** that can be scheduled by an operating system.

Threads allow us to execute different **tasks concurrently** with a minimum expense of resources since all the threads of the same process **share** the memory space and its creation consumes little processor time. They are lightweight processes.

In the same way that an operating system allows different processes to be executed at the same time by concurrence or parallelism, within a **process** there will be one or several **threads running**.

The execution of a **process** starts with a **single thread**, but more threads can be created on the fly.

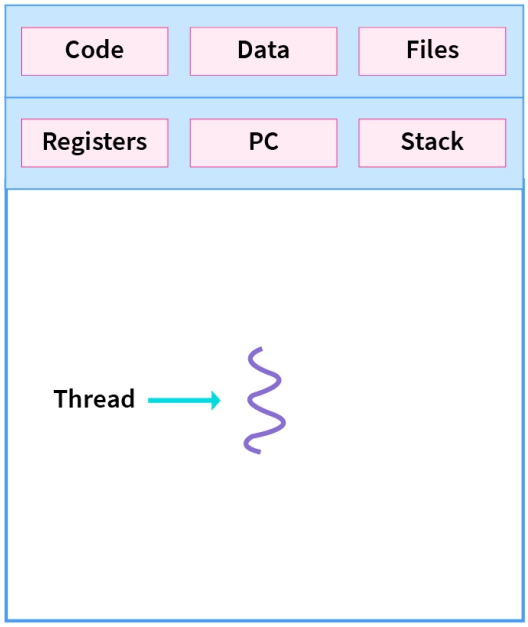
Different threads of the same process share:

- The memory space allocated to the process
- File access information (including stdin, stdout and stderr)

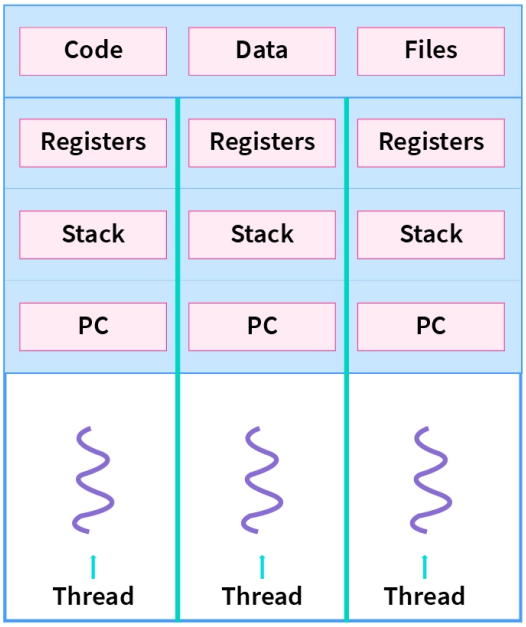
The above characteristics are what differentiate them from processes. On the other hand, each thread has its own values for:

- The processor registers
- The state of its stack, where, among other things local variables are stored

Therefore, we are going to use **threads** to perform **concurrent** and/or **parallel programming** within a **process**.



Single-threaded process



Multithreaded process

Information exchange between threads is straightforward, since the different threads of the same process share the memory allocated to the process.

However, **threads** must **coordinate** for access to memory contents and files, which makes this **coordination** and **synchronisation** the **tricky** part to use.

When a thread is invoked, there will be two paths of execution. One path will execute the thread and the other path will follow the statement after the thread invocation. There will be a separate stack and memory space for each thread.

Thread creation in Java

When a **Java program** is launched (becomes a process) it starts running through its **main()** method which is executed by the **main thread**, a **special thread** created by the **Java VM** to run the application. From a **process,** you can create and start as **many threads** as you need. These threads will execute parts of the application code in parallel with the main thread.

A thread created by the main() method can create other threads, which at the same time can create new threads. The threads created depend directly on their creators, thus creating hierarchies of parents and children of threads.

Threads in Java are objects like any other. A **thread** is an instance of the **java.lang.Thread** class, or instances of classes that inherit from it. As we have already mentioned, in addition to being objects, threads can execute code.

The most commonly used way to tell a thread what code we want it to execute is by creating a class that implements the **java.lang.Runnable** interface.

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

After creation, a **thread** can be **launched** with its **start()** method.

Whatever the thread must do, it must be included in the implementation of the run method. We have **three possibilities** to implement such an interface:

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

Java class implementing the Runnable interface

The first way we are going to see is by **creating** a class that implements the interface Runnable. We can see a basic example in the following code:

```
public class MyRunnable implements Runnable {

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

All the implementation does is print the text "MyRunnable thread running". After executing that line of code, the run method terminates and the thread that was executing it would stop.

To **test** the code and **launch** two **threads**, we can use the following code:

```
public class LaunchThreads {
    public static void main(String[] args) {
        Thread th1=new Thread(new MyRunnable());
        Thread th2=new Thread(new MyRunnable());
        th1.start();
        th2.start();
        System.out.println("Main thread finished");
    }
}
```

Anonymous class implementation of the Runnable interface

Another way to get an object that implements Runnable is to create an anonymous class. Here is an example of how to do this:

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

Except that it uses an anonymous class, the example does exactly the same as the previous example in which a class was created that implemented the interface.

Implementation of Runnable through a Lambda expression

For the third way, we are going to rely on the characteristic of the Runnable interface, that is, that it only has a single method to implement, the run method. Runnable is a functional interface so we can create a lambda expression that will not give rise to confusion about the method to be executed. Let's look at an example:

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

The Thread class

In addition to implementing the **Runnable interface**, other way to tell a thread what code to execute is to create a subclass of **java.lang.Thread** and **override the run()** method. The **Thread class** implicitly **implements** the **Runnable interface**. As with Runnable, the **run()** method contains the code that a thread will execute when the **start()** method is called.

Let's see an example of creating a class that inherits from the Thread class:

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

To create and launch a new thread, we must use the following code:

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

If we directly call the **run()** method then no new thread will be created and run() method will be executed as a normal method call on the current calling thread itself and **no multi-threading** will take place.

We can't call the **start()**method **twice** otherwise it will throw an **IllegalStateException**.

The call to the **start()** method returns control to the main thread as soon as the associated thread starts. When we call the **start()** method, the main thread does not wait for the **run()** method to execute completely before continuing. The run() method will be executed in a different thread, possibly by a different processor, entering the process queue to compete for processing units in the system. As in the previous cases, when the run() method executes, it will display the message "MyThread running" and the thread will terminate its execution (and its life) because the run() method code is terminated.

The example can be repeated with an anonymous class, but no longer with a lambda expression, as the Thread class has many more methods and is not a functional interface.

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

thread.start();
```

The example will display the message "Thread running" when the run() method is executed by the new thread.

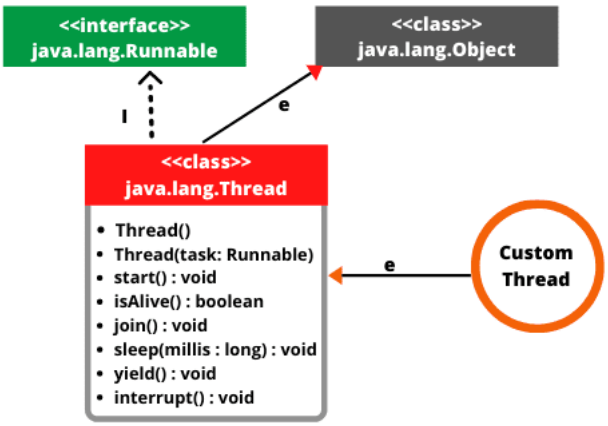
Termination of a process

In a **single-threaded** process, we are used to the fact that the process is still running (alive) as long as the code we have put in the main thread is running. Specifically, as long as the **main-thread** is running.

When a **process** has more **threads**, the rule is that the **process** does not end its execution until the **last** of the **threads** has **finished**. So, we may find that the main-thread ends and the process continues to run.

Thread or Runnable?

There is nothing to indicate that one way is better than the other. Both methods are similar, and the result is the same. The **preferred** method should be to **implement Runnable** and pass the instance to the Thread constructor.

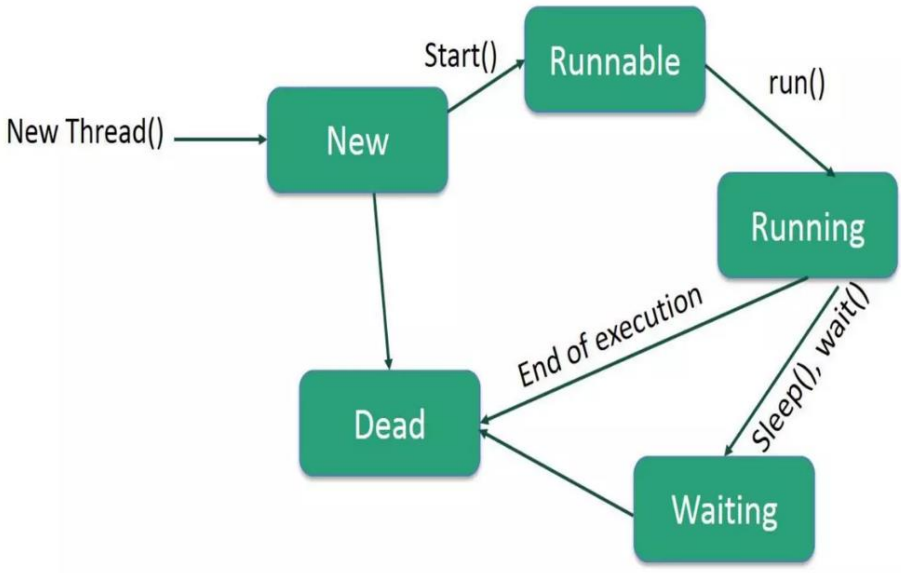


A few reasons against using **Thread**:

- When we inherit from the Thread class, we are not overriding any of its methods. Instead, we are overriding a method of the Runnable interface (which Thread implements internally). This is a clear violation of the Thread IS-A principle.
- When we pass the Runnable instance and use it as an argument in the Thread constructor, we are using composition and not inheritance, which allows much more flexibility.
- If we inherit from Thread, we can no longer inherit from other classes. This is a big problem when using libraries or graphical components, as Java does not allow multiple inheritance.
- From Java 8 onwards, the Runnable interface can be represented with lambda expressions.

Life cycle of a thread in Java

The life cycle of a thread in Java is controlled by the JVM. The Java thread states are as follows: new, runnable, running, waiting and terminated.



New. A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable. After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting. Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing. Other times, a thread enters the waiting state for a specified interval of time. A thread transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (dead). A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread scheduling

Execution of multiple threads on a single CPU, in some order is called scheduling.

In general, the runnable thread with the highest priority is active (running).

Java is **priority-preemptive** so, if a high-priority thread wakes up, and a low-priority thread is running, then the high-priority thread gets to run immediately.

Methods of the java.lang.Thread class

Some of the most commonly used methods of the Thread class are the following:

Method	Description
void start()	Makes a new thread execute the code of the run() method.
boolean isAlive()	Checks if a thread is alive or not.
static void sleep(long ms)	Static method that changes the state of the thread to locked for the specified milliseconds.
void run()	This is the code that the thread executes. It is called by the start() method. It represents the life cycle of a thread.
String toString()	Returns a readable representation of a thread [name, priority, group_name].
long getId()	Returns the identifier of the thread (this is an id assigned by the process).
static void yield()	Makes the thread stop running at the moment, going back to the queue and allowing other threads to be executed.

Method	Description
void join()	It is called from another thread and causes the invoking thread to block until the thread terminates. It is similar to waitFor() for the processes.
String getName()	Gets the name of the thread.
String setName(String name)	Change the name of the thread.
int getPriority()	Gets the priority of the thread.
void setPriority(int p)	Modify the thread priority.
void interrupt()	Interrupts the execution of the thread, causing an InterruptedException to be thrown.
boolean interrupted()	Checks if a thread has been interrupted.
Thread currentThread()	Static method of the Thread class that returns a reference to the thread that is executing the code.
boolean isDaemon()	Checks if a thread is a service/daemon. A low-priority process/thread that runs independently of its parent process. A process can terminate even if a daemon thread is still running.
void setDaemon(boolean on)	Turns a thread into a daemon/service. By default, all threads are created as user threads.
int activeCount()	Returns the number of threads belonging to a group that are still active.
Thread.State getState()	Returns the current status of the thread. Possible values are NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING or TERMINATED.

The Thread class also has about **9 constructors**, most of them are duplicated allowing to receive a Runnable object as a parameter.

Thread class constructors

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

Thread(ThreadGroup group, String name)
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target)
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)

Let's look at a **practical example** of the use of all these methods.

First, we create an InfoThread class that implements the Runnable interface and shows several information about the launched thread.

```
public class InfoThread implements Runnable{
    @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());
    }
}
```

To launch the thread several times, we create the following implementation:

```
public class LaunchSeveralInfoThread {
    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 Thread((i%2==0)?even:odd, new InfoThread(),"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");
    }
}
```

If you run the program, you will see that although the threads are launched in order (1, 2, 3 etc.) their execution is no longer sequential. They may not be executed sequentially, i.e. thread 1 may not be the first to write its name to System.out. This is because, in principle, threads are executed in parallel and not sequentially. The JVM and/or the operating system determine the order in which the threads are executed. This order does not necessarily have to be the same as the order in which they were started.

Pausing a thread

A thread can pause its own execution by calling the static method **Thread.sleep()**. The sleep() method takes as a parameter the number of milliseconds it wants to be paused before it returns to ready for execution. It is not a 100% accurate method (even less if we use the version that receives ms and ns), but it is still quite accurate. Here is an example of a thread pausing for 3 seconds (3000ms) by calling the sleep() method:

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

Thread.sleep() interacts with the thread scheduler to put the **current thread** in a **wait state** for a specified period of time. Once the wait time is over, the thread state is changed to a runnable state and waits for the CPU for further execution. The actual time that the current thread sleeps depends on the thread scheduler that is part of the operating system.

Thread.sleep() is a simple way to introduce **delays** without being **busy waiting**.

Thread priority management

Threads inherit the priority from the parent in Java, but this value can be changed with the **setPriority()** method and with **getPriority()** we can know the priority of a thread.

Values of **priority** are between **1** and **10**. The **higher** the **value**, the **higher** the **priority**. The Thread class defines the following constants: **MIN_PRIORITY** (value 1) **MAX_PRIORITY** (value 10) and **NORM_PRIORITY** (value 5). The scheduler chooses the thread according to its priority. If two threads have the same priority, it performs a **round-robin**, that is to say, it alternates the threads in a cyclical way.

Creating threads using the Callable interface

There are two ways of creating threads, one by **extending** the **Thread** class and other by creating a thread with a **Runnable**. However, one feature lacking in Runnable is that we cannot make a thread return a result when it terminates (when run() completes). For supporting this feature, the Callable interface is present in Java. Besides, the Runnable interface does not allow to throw any checked exceptions (exceptions caught at compile time).

The **Callable interface** in Java has a single method **call()** that can **return** any **object**.

When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned.

How will the program store and obtain this result later? For this, a **Future object** can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a **Future** is basically one way the main thread can **keep track** of the **progress** and **result** from other threads.

Observe that **Callable** and **Future** do two different things – **Callable** is like Runnable, in that it **encapsulates** a **task** that is meant to **run** on **another thread**, whereas a **Future** is used to **store** a **result** obtained from a **different thread**.

Method	Description
public boolean cancel (boolean mayInterrupt)	Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true.
public Object get() throws InterruptedException, ExecutionException	Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result.
public boolean isDone()	Returns true if the task is complete and false otherwise.

To create the thread, a Runnable is required. To obtain the result, a Future is required.

The Java library has the concrete type **FutureTask**, which **implements Runnable** and **Future**, combining both functionalities conveniently. A **FutureTask** can be created by providing its constructor with a **Callable**. Then the **FutureTask** object is provided to the constructor of Thread to create the Thread object. Thus, indirectly, the thread is created with a Callable. For further emphasis, note that there is no way to create the thread directly with a Callable.

Below are the steps to implement the **Callable** interface in Java

1. Create a class that implements the **Callable** interface.
2. Override the **call()** method of the **Callable** interface in the class.

3. Create a **FutureTask** object by passing the class object that implemented the **Callable** interface.
4. Create a **Thread** object in the main class by passing the **FutureTask** object created.
5. Call the **start()** method of the **Thread** object.
6. Get the result returned by the callable class using the **get()** method of the **FutureTask** object.

The **call()** method of the Callable interface can throw both checked and unchecked (exceptions occurring at runtime) exceptions.

The computation can be cancelled using the **cancel()** method of the **FutureTask** object.

Let's observe the following code snippet which implements the **Callable interface** and returns a **random number** ranging from **0 to 9** after making a delay between **0 to 9 seconds**.

```
public class JavaCallable implements Callable {
    @Override
    public Object call() throws Exception {
        // Creating an object of the  Random class
        Random randObj = new Random();

        // generating a random number between 0 to 9
        Integer randNo = randObj.nextInt(10);

        // the thread is delayed for some random time
        Thread.sleep(randNo * 1000);

        // return the object that contains the
        // generated random number
        return randNo;
    }
}
```

The following code produces 10 different threads. Each thread invokes the **call()** method, generates a **random number**, and returns it. The **get()** method is used to **receive** the returned random number object obtained from the different threads to the main thread. The **get()** method is declared in the **Future** interface and implemented in the **FutureTask** class. A **Future** represents the result of an asynchronous computation.

```
public class Main {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // FutureTask is the concrete class
        // creating an array of 10 objects of the FutureTask class
        FutureTask[] randomNoTasks = new FutureTask[10];

        // loop for spawning 10 threads
        for (int j = 0; j < 10; j++) {
            // Creating a new object of the JavaCallable class
            Callable callable = new JavaCallable();
            // Creating the FutureTask with Callable
            randomNoTasks[j] = new FutureTask(callable);
            // Since FutureTask implements Runnable,
            // one can create a Thread with a FutureTask object
            Thread th = new Thread(randomNoTasks[j]);
```

```
            th.start();
        }
        // loop for receiving the random numbers
        for (int j = 0; j < 10; j++) {
            // invoking the get() method
            Object o = randomNoTasks[j].get();
            // The get method holds the control until the result is received
            // The get method may throw the checked exceptions
            // like when the method is interrupted. Because of this reason
            // we have to add the throws clause to the main method
            // printing the generated random number
            System.out.println("The random number is: " + o);
        }
    }
}
```