 MINISTRY FOR EDUCATION AND SCIENCE OF RUSSIA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### SAINT PETERSBURG ELECTROTECHNICAL UNIVERSITY

#### «LETI»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

APPROVED

\_\_\_\_\_\_\_\_\_

10.06.2020

Date

<The Java™ Tutorials>

***FinalReport***

**Realized by:**

**Drama Houcine**

**Framed by:**

**Natalya Razmochaeva**

        Electronic&Paper /Электронныйибумажный

вид носителя данных

         24pages / страниц

объём документа

St. Petersburg / Санкт-Петербург

2020

**The Java programming language tutorial / course**

1. **Introduction and history :**

* **How did we get to Java?**

The history of the Java programming language begins in 1991. At that time, the company Sun Microsystems sought to propose a new programming language (although strongly inspired by existing languages, and in particular C ++) allowing the development of portable applications for embedded systems (and more specifically household appliances) as well as an execution environment allowing the execution of the programs produced from their new language. The motivation of this new project (Green Project) is to be able to make household appliances better controllable, more interactive but also to allow communication between these devices. All of these devices need to be controlled via a universal remote control called \* 7 (pronounce Star 7). This remote control was based on the execution environment (a mini operating system) allowing the execution of programs and communication with all devices.

As we have just said, this new language, initially called Oak (Oak meaning oak), was inspired in particular by C ++. However, it is relatively complex and above all not very portable (notably because of the size, not imposed, of the data types of the language). James Gosling, one of the players in the project (now considered the father of Java) therefore decided to create an object-oriented language incorporating the main characteristics of the C ++ language, while modernizing its use (removing the pre-processor, strengthening typing control , ...). One of the most important points resides in the fact that the size of the types of data is now imposed: this point making it possible to make the programs more portable in order to simplify their integration in any type of material (processors 16 bits, 32 bits , ...). Another characteristic of this language, and this from its creation, was that it allowed the generation of machine code that could be executed on any type of platform: this feat was possible thanks to the incorporation an emulator in the runtime environment.

Unfortunately the Sun Microsystems team had to resolve to change the name of their language, the Oak name already being used: the new name chosen will be ... Java, of course. The "legend" Java wants that this name was chosen in honor of the favorite drink of the programmers of the team: namely a coffee arabica from the island of Java. Some of you will better understand certain logos and terminologies related to Java. I am thinking in particular of Java Beans technology whose icons most often used are coffee beans (by the way, the English word bean meaning seed, of course coffee).

The problem is, this project was not as successful as it should have been and, from a certain point of view, Java could have disappeared. However, at the same time, the Web appeared. The web allowed machines connected via the Internet to exchange hypertext documents via HTTP (HyperText Transfer Protocol) and HTML (HyperText Markup Language). These machines were (and still are) based on heterogeneous hardware and used multiple operating systems. How to enrich the rendering of Web documents of the time given this diversity of system? Java of course had all the characteristics that made it an ideal language for the Web. In 1994, Sun Microsystems developed the HotJava web browser. This integrates a Java execution environment allowing to insert into HTML pages small Java applications: the famous Java applets. Netscape quickly integrated Java technology into their browser and by the mid-90s Java had the tremendous boom that everyone knows about it today.

In 2009, Oracle Corporation bought Sun Microsystems and thus took control of Sun technologies, especially Java. James Gosling will eventually leave Oracle in 2010, but Java continues to develop.

* **Main features of the Java language :**

This language is a real programming language: it integrates almost all that we know how to do best in terms of programming language, while removing what has proven to be sources of errors or difficulties for the developer . A first characteristic of the Java language is that it is fully object oriented: the basic building block of the program is therefore the object, an instance of a class (we will explain all this in detail in the chapter concerned).

Another of the main characteristics of this language is that the Java code will be compiled for a so-called virtual machine: the resulting machine code is called "Byte Code". During execution, the machine code produced will be transformed into a machine code understandable by the microprocessor that you use. The reason for this choice is that Java wants to be portable: the programmer no longer wishes to make adaptations so that his program can run on such or such machine. If this choice has advantageous points, know that there is also one which is much less: converting an instruction of the JVM (Java Virtual Machine) into an instruction understandable by the machine where the program is executed takes time. time. To overcome this problem, two solutions have been proposed. The first is to include in the compiler the possibility of compiling native code (for a specific machine): but this solution is not supported by SUN technologies. The second is more complex: the JVM includes a JIT (Just In Time compiler). The JIT is responsible for translating the intermediate machine code (for the JVM) directly into native code (this can be done the first time you use the class). This translation phase certainly takes a little time, but once it is done, and this until the end of the program execution, no more code translations (on the class considered) will be redone. This technique considerably improves the speed of execution of programs coded in Java.

Also note that there are Java processors. These directly run a Java program without the need for an emulator. Although this solution may seem interesting, it is rarely implemented. The few cases of using a Java processor are often linked to embedded systems. Indeed, apart from the time necessary for the transformation of the intermediate code into native machine code (but this is done once per class when loading it) a Java processor will (today) bring nothing more compared to an emulator equipped with a JIT.

The Java library is very rich: it is made up of several thousand classes, themselves often made up of dozens of methods (of functions if you prefer). We will come back to these terminology points later. For now, just note that we can do almost everything in Java: access to databases, manipulation of XML files, graphics, distributed applications, Web applications, ...

Memory management is finally no longer the responsibility of the programmer: in fact, automated memory deallocation techniques have existed for many years, they are very effective and only a few languages ​​integrate them (at least at the time ). A garbage collector was therefore added to the execution environment. Over the course of the program, you allocate the memory you need. But you don't release it: there is no equivalent to free (C language) or delete (C ++ language).

In Java, there is no macro-code generation mechanism, as is the case in C or C ++, as well as for other languages. The syntax, as well as some semantic points, are inspired by C ++ and therefore by C. This last point is in our opinion a good thing: these languages ​​are now established, they have proven themselves and now have many followers of their very famous operators (variable ++; for example). But this opinion is not necessarily shared when we discover the syntax for the first time. Java is also multithreaded, distributed, robust and secure (many things have been done on this point), ...

* **Java language distribution mode :**

All the resources necessary for developing Java applications are accessible from the Oracle site: http://www.oracle.com/technetwork/java/index.html. You will find on this site Java runtime environments (for the main known platforms), documentation on the Java API (the class library), Java tutorials (in English), discussion forums, .. Everything is free to download.

In addition, since 2007, Java has gone into "Open Source": this means that the source code of the Java runtime environment is available on the Internet and that you can contribute (in different ways) to the future development of Java. For further information on this point, I refer you, in particular, to the site http://openjdk.java.net/.

Several Java platforms are available for download. Here is some additional information on these platforms:

* **Java SE (Java Standard Edition) also called JDK:** this distribution is used to develop Java applications that will run directly on a client computer. It comes in two main forms:
* **The JRE (Java Runtime Environment):** it's just the execution environment. It is necessary for the execution of a Java program on a client computer.
* **The JDK (Java Development Kit):** it is a JRE supplemented by a certain number of development tools (compilers in particular) and of profiling of Java applications.
* **Java EE (Java Enterprise Edition) / Jakarta EE:** a complement to Java SE allowing the development of enterprise applications. We deploy the applications developed via this standard on a J2EE compatible application server (Web Sphere, JBoss, Web Logic, ...).
* **Java ME (Java Micro Edition):** a "light" distribution package for embedded platforms (mobile telephony, for example).
* **The different versions of Java SE :**

You should know that the versions of Java are specified by the [JCP (Java Community Process).](https://jcp.org/en/home/index) Although under Oracle governance, the JCP brings together people from different companies. You can view [this link](https://jcp.org/en/participation/members) for a list of current JCP members. These people write JSR (Java Specification Request) which specify the different aspects of Java technology.

Regarding Java SE, all the specifications (JSR) of this platform are included on this page: [The Java SE specifications](https://docs.oracle.com/javase/specs/). I strongly invite you, when you have finished this tutorial on Java, to consult these documents which are very enriching.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Java SE version | Release date | Code name | JSR  (Java Specification Request) | Number of types  (offered by the standard API) |
| JDK 1.0 | January 23, 1996 | Oak |  | 211 |
| JDK 1.1 | February 19, 1997 |  |  | 477 |
| J2SE 1.2 | December 9, 1998 | Playground |  | 1524 |
| J2SE 1.3 | May 8, 2000 | Kestrel | [JSR 58](https://jcp.org/en/jsr/detail?id=58) | 1840 |
| J2SE 1.4 | February 6, 2002 | Merlin | [JSR 59](https://jcp.org/en/jsr/detail?id=59) | 2723 |
| Java SE 5.0 | September 30, 2004 | Tiger | [JSR 176](https://jcp.org/en/jsr/detail?id=176) | 3270 |
| Java SE 6.0 | December 11, 2006 | Mustang | [JSR 270](https://jcp.org/en/jsr/detail?id=270) | 3777 |
| Java SE 7.0 | July 7, 2011 | Dolphin | [JSR 336](https://jcp.org/en/jsr/detail?id=336) | 8000 |
| Java SE 8.0 (LTS) | March 2014 | Spider | [JSR 337](https://jcp.org/en/jsr/detail?id=337) |  |
| Java SE 9.0 | September 21, 2017 |  | [JSR 379](https://jcp.org/en/jsr/detail?id=379) |  |
| Java SE 10 | March 20, 2018 |  | [JSR 383](https://jcp.org/en/jsr/detail?id=383) |  |
| Java SE 11 (LTS) | September 26, 2018 |  | [JSR 384](https://jcp.org/en/jsr/detail?id=384) |  |
| Java SE 12 | March 19, 2019 |  | [JSR 386](https://jcp.org/en/jsr/detail?id=386) |  |
| Java SE 13 | September 17, 2019 |  | [JSR 388](https://jcp.org/en/jsr/detail?id=388) |  |
| Java SE 14 (LTS) | March 17, 2020 |  | [JSR 389](https://jcp.org/en/jsr/detail?id=389) |  |

* **Other equivalent solution on the market :**

The main competitor of Java is none other than the Microsoft .NET framework. In fact the two solutions are extremely close. You can get the framework and find a lot of additional information from the following internet address: [http://msdn.microsoft.com](http://msdn.microsoft.com/). This is a site for Microsoft developers only. Its access is free.

1. **Difference between Process and Thread :**

* [**Process**](https://www.geeksforgeeks.org/gate-notes-operating-system-process-management-introduction/)**:**  
  Process means any program is in execution. Process control block controls the operation of any process. Process control block contains the information about processes for example: Process priority, process id, process state, CPU, register etc. A process can creates other processes which are known as Child Processes. Process takes more time to terminate and it is isolated means it does not share memory with any other process.
* [**Thread**](https://www.geeksforgeeks.org/operarting-system-thread/)**:**  
  Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. A thread have 3 states: running, ready, and blocked.

Thread takes less time to terminate as compared to process and like process threads do not isolate.

# Thread Objects :

# A user-mode thread object represents a path of execution within the current process. Every user-mode thread object is implemented through the use of an embedded kernel-mode thread object.

# A kernel-mode thread object is an instance of a kernel-defined dispatcher object type. The thread that it represents is the basic schedulable entity in the operating system.

A thread object:

* Is dispatched for execution by the kernel.
* Has the following properties at any given moment:
  + dispatch state
  + priority
  + context
  + Execution mode (kernel or user)
  + affinity
* Is "owned by" a process object but can attach itself to another process's address space.

Usually, most drivers execute in the context of the currently running thread, that is, in an arbitrary thread context. While a file system driver can create an independent process for its own device-dedicated threads, file systems usually avoid setting up a driver-created process and threads in order to conserve system memory and to avoid the overhead of context switches.

FSs (and other drivers) can set up device-dedicated (system-process) threads and/or FSs can use system worker threads if they need a driver-specific thread context in which to execute. Drivers use the kernel-mode **PsXxx** routines to create processes and/or device-dedicated threads. FSs call **ExXxx** routines to use system worker threads.

1. **definition of thread :**

A thread is a unit that is part of a program. This unit works independently and in parallel with other threads. The main advantage of threads is to be able to distribute different treatments of the same program into several distinct units to allow their "simultaneous" executions.

On a single-processor machine, it is the operating system which allocates time to use the CPU to perform the processing of each thread, thus giving the impression that these processing operations are carried out in parallel.

On a multiprocessor machine, the operating system can distribute the execution on several cores, which can effectively allow processing to be carried out in parallel.

Depending on the operating system and the JVM implementation, threads can be managed in two ways:

* match a native system thread
* correspond to a thread managed by the virtual machine

In both cases, this has no impact on the code which remains the same.

The JVM itself creates several threads for its own needs: the application execution thread, one or more threads for garbage collection, ...

The java.lang.Thread class and the java.lang.Runnable interface are the bases for developing threads in java.

The operating system will have to allocate processing time for each thread on the CPU (s) of the machine. The more threads, the more the system will have to switch. In addition, a thread requires resources to execute, in particular a memory space called stack. It is therefore necessary to control the number of threads that are launched in the same JVM.

However, using multiple threads generally improves performance, especially if the machine has multiple cores, since in this case multiple threads can really run in parallel. It is also frequent that the processing of a thread is waiting for a resource: the system can then more quickly allocate CPU time to other threads which are not.

The use of the Thread class is fairly low level. From Java 5, the java.util.concurrency package offers higher-level functionalities to facilitate the implementation of parallel processing and improve the performance of concurrent access management.

## Thread.sleep in Java :

Thread.sleep() method can be used to pause the execution of current thread for specified time in milliseconds. The argument value for milliseconds can’t be negative, else it throws IllegalArgumentException.

There is another overloaded method sleep(long millis, int nanos) that can be used to pause the execution of current thread for specified milliseconds and nanoseconds. The allowed nano second value is between 0 and 999999.

### Java Thread Sleep Example :

### Here is a simple program where Thread.sleep() is used to pause the main thread execution for 2 seconds.

### ThreadSleep.java

package com.journaldev.threads;

public class ThreadSleep {

public static void main(String[] args) throws InterruptedException {

long start = System.currentTimeMillis();

Thread.sleep(2000);

System.out.println("Sleep time in ms = "+(System.currentTimeMillis()-start));

}

}

If you will run the above program, you will notice that the thread sleep time it prints is slightly greater than 2000. This is caused by how thread sleep works and operating system specific implementation of thread scheduler.

### Java Thread Sleep important points :

* It always pause the current thread execution.
* The actual time thread sleeps before waking up and start execution depends on system timers and schedulers. For a quiet system, the actual time for sleep is near to the specified sleep time but for a busy system it will be little bit more.
* Thread sleep doesn’t lose any monitors or locks current thread has acquired.
* Any other thread can interrupt the current thread in sleep, in that case InterruptedException is thrown.

### How Thread Sleep Works :

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

# Interrupts :

[Interrupt](https://www.geeksforgeeks.org/purpose-of-an-interrupt-in-computer-organization/) is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high priority process requiring interruption of the current working process. In I/O devices one of the bus control lines is dedicated for this purpose and is called the *Interrupt Service Routine (ISR)*.

When a device raises an interrupt at lets say process i, the processor first completes the execution of instruction i. Then it loads the Program Counter (PC) with the address of the first instruction of the ISR. Before loading the Program Counter with the address, the address of the interrupted instruction is moved to a temporary location. Therefore, after handling the interrupt the processor can continue with process i+1.

# Interrupting a Thread:

While the processor is handling the interrupts, it must inform the device that its request has been recognized so that it stops sending the interrupt request signal. Also, saving the registers so that the interrupted process can be restored in the future, increases the delay between the time an interrupt is received and the start of the execution of the ISR. This is called Interrupt Lattency.

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

## The 3 methods provided by the Thread class for interrupting a thread :

* public void interrupt()
* public static boolean interrupted()
* public boolean isInterrupted()

# The join() method :

# The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

# The SimpleThreads Example :

### Java Thread Example – implementing Runnable interface :

To make a class runnable, we can implement java.lang.Runnable interface and provide implementation in public void run() method. To use this class as Thread, we need to create a Thread object by passing object of this runnable class and then call start() method to execute the run() method in a separate thread.

Here is a java thread example by implementing Runnable interface.

package com.journaldev.threads;

public class HeavyWorkRunnable implements Runnable {

@Override

public void run() {

System.out.println("Doing heavy processing - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Doing heavy processing - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

### Java Thread Example – extending Thread class :

We can extend **java.lang.Thread** class to create our own java thread class and override run() method. Then we can create it’s object and call start() method to execute our custom java thread class run method.

Here is a simple java thread example showing how to extend Thread class.

package com.journaldev.threads;

public class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

@Override

public void run() {

System.out.println("MyThread - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("MyThread - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

# Synchronization :

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive :

* Synchronized method.
* Synchronized block.
* static synchronization.

1. Cooperation (Inter-thread communication in java)

### Mutual Exclusive :

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

### Concept of Lock in Java :

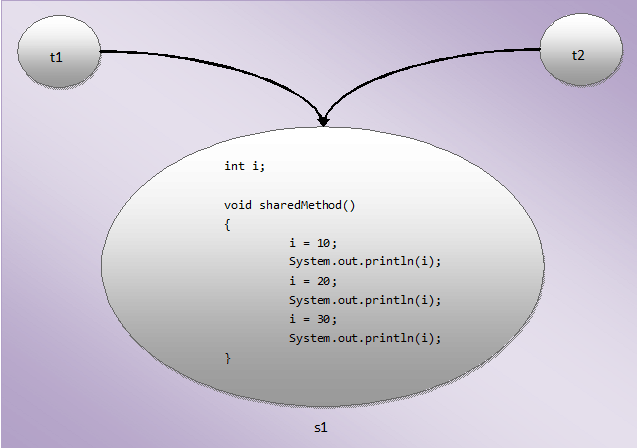
Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

1. **Thread interference :**

**Thread interference in java** is a condition which occurs when more than one threads, executing simultaneously, access same piece of data. When more than one threads have access to same data, it is possible that data may get ; corrupted or one may not get the desired output. Thread interference occurs when code written is not **thread safe**.

This is an example of **Thread Interference**. Thread interference occurs when sequence of steps of more than one threads overlap. You can follow the Oracle documentation of thread interference [here](http://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html). The above example can be described by the diagram as below.

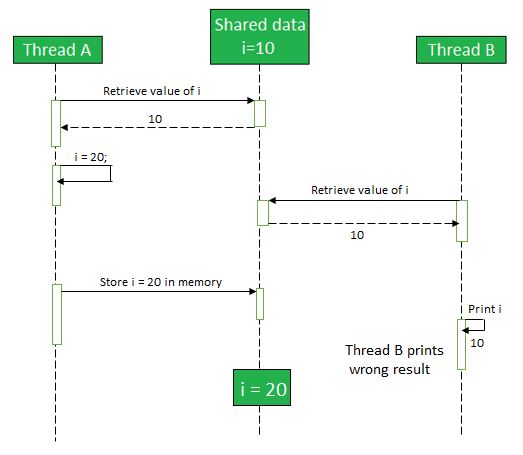


# Memory Consistency Errors :

In multithreading, there can be possibilities that the changes made by one thread might not be visible to the other threads and they all have inconsistent views of the same shared data. This is known as memory consistency error.

Memory consistency is more of an architecture-based concept than Java-based. Accesses to main memory might not occur in the same order in which the CPU initiated them, especially for write operations which often go through hardware write buffers so that the CPU need not wait for them. CPUs guarantee that the order of writes to a single memory location is maintained from the perspective of all CPUs, even if CPUs perceive the write time of other CPUs differently than the actual time. This sometimes leads to memory inconsistency due to lack of visibility of the correct data.

* **Sequence diagram for memory consistency error :**



# Intrinsic Locks and Synchronization :

In Java, an intrinsic lock is implied by each use of the synchronized keyword. In this case, the locking is performed by Java behind the scenes. (This is distinct from the programmer using or defining an explicit lock object themselves.)

Each use of the synchronized keyword is associated with one of the two types of intrinsic lock:

* an "instance lock", attached to a single object
* a "static lock", attached to a class

If a method is declared as synchronized, then it will acquire either the instance lock or the static lock when it is invoked, according to whether it is an instance method or a static method.

The two types of lock have similar behaviour, but are completely independent of each other.

Acquiring the instance lock only blocks other threads from invoking a synchronized instance method; it does *not* block other threads from invoking an *un-synchronized* method, nor does it block them from invoking a *static* synchronized method.

Similarly, acquiring the static lock only blocks other threads from invoking a static synchronized method; it does *not* block other threads from invoking an *un-synchronized* method, nor does it block them from invoking a synchronized *instance* method.

Outside of a method header, synchronized(this) acquires the instance lock.

The static lock can be acquired outside of a method header in two ways:

* synchronized(Blah.class), using the class literal
* synchronized(this.getClass()), if an object is available

# Atomic Access :

Variables shared between multiple threads (e.g., instance variables of objects) have atomic assignment guaranteed by the Java language specification for all data types except for longs and doubles. Actually, the storing of a value into a variable takes two primitive operations, a *store* and a *write*. However, the language specification also states that once a store operation occurs on a particular variable, no other store operation is allowed on that variable until the write operation has occurred. The specification allows longs and doubles to be stored in two separate sets of store+write operations, hence their exception to atomicity. A similar atomic specification applies for reading variables. This means that access and update of variables are automatically synchronized (as long as they are not longs or doubles). If a method consists solely of a variable access or assignment, there is no need to make it synchronized for thread safety, and every reason not to do so for performance. Thread safety extends further to any set of statements that are accessing or assigning to a variable independently of any other variable values. The exclusion here precludes setting a variable that depends on the value of another variable as being thread-safe; this would be two separate operations, which is inherently not thread-safe. For example:

public void setMe(Object o) {me = o;}

public Object getMe( ) {return me;}

# Deadlock and Starvation and Livelock:

# A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

# Deadlock: A *deadlock* is a state in which each member of a group of actions, is waiting for some other member to release a lock. A *livelock* on the other hand is almost similar to a deadlock, except that the states of the processes involved in a livelock constantly keep on changing with regard to one another, none progressing. Thus Livelock is a special case of resource starvation, as stated from the general definition, the process is not progressing.

# Starvation: Starvation is a problem which is closely related to both, Livelock and Deadlock. In a dynamic system, requests for resources keep on happening. Thereby, some policy is needed to make a decision about who gets the resource when. This process, being reasonable, may lead to a some processes never getting serviced even though they are not deadlocked.

# Livelock:

# var l1 = .... // lock object like semaphore or mutex etc

# var l2 = .... // lock object like semaphore or mutex etc

# // Thread1

# Thread.Start( ()=> {

# while (true) {

# if (!l1.Lock(1000)) {

# continue;

# }

# if (!l2.Lock(1000)) {

# continue;

# } /// do some work });

# // Thread2

# Thread.Start( ()=> {

# while (true) {

# if (!l2.Lock(1000)) {

# continue;

# }

# if (!l1.Lock(1000)) {

# continue;

# }

# // do some work

# });

**Livelock** occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

# https://media.geeksforgeeks.org/wp-content/uploads/aaa-1.png

# Immutable Objects :

Immutable objects are objects that don't change. You make them, then you can't change them. Instead, if you want to change an immutable object, you must clone it and change the clone while you are creating it.

A Java immutable object must have all its fields be internal, private final fields. It must not implement any setters. It needs a constructor that takes a value for every single field.

Immutable objects come in handy in multi-threaded environments and in streams. It is great to rely on objects not changing mid-stream. Bugs caused by a thread changing another thread's object are often subtle and are very, very hard to track down. Immutable objects stop these whole class of problems in their tracks.

# A Strategy for Defining Immutable Objects :

A Strategy for Defining Immutable Objects in JAVA.

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1.Don't provide "setter" methods — methods that modify fields or objects referred to by fields.

2.Make all fields final and private.

3.Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.

4.If the instance fields include references to mutable objects, don't allow those objects to be changed:

4.1. Don't provide methods that modify the mutable objects.

4.2. Don't share references to the mutable objects.Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

01. import java.util.Date;

02.

03. /\*\*

04. \* Always remember that your instance variables will be either mutable or immutable.

05. \* Identify them and return new objects with copied content for all mutable objects.

06. \* Immutable variables can be returned safely without extra effort.

07. \* \*/

08. public final class ImmutableClass

09. {

10.

11. /\*\*

12. \* Integer class is immutable as it does not provide any setter to change its content

13. \* \*/

14. private final Integer immutableField1;

15.

/\*\*

16. \* String class is immutable as it also does not provide setter to change its content

17. \* \*/

18. private final String immutableField2;

19.

/\*\*

20. \* Date class is mutable as it provide setters to change various date/time parts

21. \* \*/

22. private final Date mutableField;

23.

24. //Default private constructor will ensure no unplanned construction of class

25. private ImmutableClass(Integer fld1, String fld2, Date date)

26. {

27. this.immutableField1 = fld1;

28. this.immutableField2 = fld2;

29. this.mutableField = new Date(date.getTime());

30. }

31.

32. //Factory method to store object creation logic in single place

33. public static ImmutableClass createNewInstance(Integer fld1, String fld2, Date date)

34. {

35. return new ImmutableClass(fld1, fld2, date);

36. }

37.

38. //Provide no setter methods

39.

40. /\*\*

41. \* Integer class is immutable so we can return the instance variable as it is

42. \* \*/

43. public Integer getImmutableField1() {

44. return immutableField1;

45. }

46.

47. /\*\*

48. \* String class is also immutable so we can return the instance variable as it is

49. \* \*/

50. public String getImmutableField2() {

51. return immutableField2;

52. }

53.

54. /\*\*

55. \* Date class is mutable so we need a little care here.

56. \* We should not return the reference of original instance variable.

57. \* Instead a new Date object, with content copied to it, should be returned.

58. \* \*/

59. public Date getMutableField() {

60. return new Date(mutableField.getTime());

61. }

62.

63. @Override

64. public String toString() {

65. return immutableField1 +" - "+ immutableField2 +" - "+ mutableField;

66. }

67. }

1. **Locking objects:**

**Object level lock**is mechanism when we want to synchronize a **non-static method**or **non-static code block** such that only one thread will be able to execute the code block on given instance of the class. This should always be done **to make instance level data thread safe.**

Every object in java has a unique lock. Whenever we are using synchronized keyword, then only lock concept will come in the picture. If a thread wants to execute synchronized method on the given object. First, it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the lock. Acquiring and release lock internally is taken care by JVM and programmer is not responsible for these activities.

# Executors :

# Threads in Java have many disadvantages essentially related to the fact that the Thread class is low level and therefore certain functionalities must be developed, for example:

# get a thread execution result

# get an exception thrown in the thread by the caller

# no thread pool is offered as standard

# wait for a set of threads to finish

# Executor Interfaces :

# The java.util.concurrent.Executor interface describes the functionalities allowing the delayed execution of tasks implemented in the form of Runnable.

# It only defines one method:

|  |  |
| --- | --- |
| Method | Role |
| void execute (Runnable command) | Execute the task provided as a parameter possibly in the future |

# Depending on the implementation, the task can be executed in a dedicated thread or in the current thread.

# It has two child interfaces:

# ExecutorService: it defines the functionality of a service allowing the execution of Runnable or Callable tasks.

# ScheduledExecutorService which inherits from the ExecutorService interface: it defines the functionalities of a service for the execution of scheduled and / or repeated tasks

# Fork/Join :

# This framework is the main novelty of the java.util.concurrent package. Like most frameworks, it is supposed to solve a problem. However, sometimes frameworks pose more problems than they solve. Since I know that your time is dear, precious reader, and that mine is too, of these frameworks, we will avoid talking about it.

# The problem he intends to solve concerns the processing of large amounts of data, digital or not. It often happens that these quantities can be treated packet by packet, each packet being able to be taken into account independently of the others. The processing of each packet provides a partial result. These results are then somehow merged into the overall treatment result.

# These packets can be processed one after the other in the same thread; let's say this is the classic way of doing things, or at least immediate.

# You can also choose to process each packet in its own thread. Once all these threads have finished their work, the results are grouped in the thread that launched them. We then need a synchronization point, which can be managed by a CyclicBarrier.

# The ForkJoinTask class models a single task. This task is sent to a thread pool, which will itself manage its execution in a thread in this pool. An elementary task can itself generate other tasks of the same type, sent to the same reserve.

# the ForkJoinPool class: manages the thread pool to which the tasks are submitted. This reserve receives tasks, and takes care of choosing which thread will execute which task. This class is designed in such a way that a very large number of small elementary tasks can be processed in a limited number of threads.

# The ForkJoinTask <V> class exposes two main methods:

# join (): returns the result of the execution of this task, type V;

# fork (): method called to launch another task in the same reserve as the one in which the current task is located.

# Concurrent Collections :

The java.util.concurrent package includes several thread-safe collections:

• ConcurrentHashMap Collection of HashMap types implementing the ConcurrentMap interface;

• CopyOnWriteArrayList collection of type ArrayList with CopyOnWrite algorithm;

• CopyOnWriteArraySet implementation of the Set interface, using CopyOnWriteArrayList as base;

• ConcurrentNavigableMap extends the NavigableMap interface;

• ConcurrentSkipListMap An analogue of the TreeMap collection with data sorting by key and with multithreading support;

• ConcurrentSkipListSet implementation of the Set interface, based on the ConcurrentSkipListMap class.

Only the first three thread-safe collections are presented on the page. As with ConcurrentNavigableMap, the iterators of this class are declared thread-safe and do not raise a ConcurrentModificationException. The ConcurrentSkipListMap class guarantees the average performance of operations for contains Key, get, put, remove and other similar methods.

# *Practical :*

# Make the corresponding change in the program below to allow the program to terminate after all threads are completed :

# importer java.util.de façon concomitante.ExecutorService;

# importer java.util.de façon concomitante.Les exécuteurs testamentaires;

# public class ExitMain1 {

# private static final int POOL\_SIZE = 50;

# public static void main(final String[] args) throws Exception {

# final ExecutorService exécuteur = Exécuteurs testamentaires.newFixedThreadPool(POOL\_SIZE);

# for (int i = 0; i < POOL\_SIZE; i++) {

# exécuteur testamentaire.soumettre(() -> Système.out.printf("Exécution de %s%n", Fil.currentThread().getName()));

# }

# exécuteur testamentaire.\*\*\*\* ;

# }

# }

# About this essay, in this code i used the function wait for waiting all the threads until its execution ends, but i got wrong answer.

# And when i revised again our essay i understand that program needs to terminate after all threads are completed its execution so i chosen the function shutdown(), and i got right answer.

2. How do you start a new Thread in Java?

Fill in the blanks

You're right!

public class StartThread {

public static void main(final String[] args) {

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

new Thread(() -> System.out.println("Started")). start();}}

Note :

To start the Java thread you will call its start() method, like this:

**thread.start();** but in our code we don’t need to use name of thread because we didn’t create specific name to our thread so we call directely method Start() ;.

**Thank You Dear Teacher**

**Natalya Razmochaeva**