# Lecture 02

## Concurrent Programming

### Concurrent execution in native Java

*Object-Oriented Concurrent Programming, 2019-2020*

*v2.3, 25-09-2019*

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

## Contents



## 1 Motivation

### Concurrent programming: What For!

**CP: What for**: Making programs potentially faster, more responsive, and more real.

- Make full use of the available hardware.

Almost all computers sold today have multiple processing cores available for computing. Although they can be used at the operating system level (attaching each one to different programs), it make little sense not to take advantage of this processing power within a single program, with the result of, probably, reducing the time required to achieve its goals.

- To make applications more responsive.

When execution of a program involves time-consuming processing (input/output, intensive calculations, etc.), a concurrent execution separated from the user interface (or other applications) makes a more responsive program.

- To meet the real world concurrency.

Real problems can rarely be described as sequential. The real world is a giant parallel "machine" in which numerous active autonomous entities interact with each other.

- Because sequential programs are a particular case of concurrent programs;
- Because it is an exciting challenge for those who like to program, and an inescapable requirement for the current and future programming needs in labor market!

## Concurrent Programming: Price to Pay!

**CP: Price to Pay!**: *Nondeterminism* of programs!

- Program design is more complex;
- Ensuring *correction* is far more difficult;
- Replication of failed tests is much more difficult;
- More complex fault management.

## Why Object Oriented Programming!

**OOP**: Good adaptation to the construction of complex programs due to its excellent *modularity* properties.

- Programs architecture based on a single kind of entity: objects;
- Allows greater abstraction in building programs: easing their understanding, and increasing their reusability, extensibility and especially their correction (specially more if used with contracts).

## Why Design by Contract!

**DbC**: It disciplines and eases the development of *programs that work*.

- Gives meaning to objects;
- Supports a disciplined, systematic "bottom-up" way to develop programs;
- Defines a clear distribution of responsibilities among the various entities of a program;
- Maximize the correctness and robustness of programs;
- Eases program debugging;
- Provides coherent documentation to code;
- Allow the construction of fault-tolerant programs.

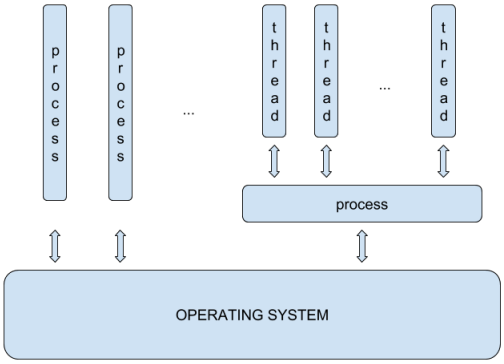


Figure 1: Processes and threads.

### Object-Oriented Concurrent Programming

**OOCP**: Add the advantages of CP and OOP (and, desirable, also DbC).

- Join the two worlds, trying to get the best of each one of them, aiming to control their problems and limitations;
- Reuse the powerful OO techniques of program construction;
- Keep the simplicity and expressiveness of objects in a concurrent context;
- Tame the inherent nondeterminism of concurrency with proper implementations of objects ensuring their contracts;

**OOCP: Challenge!**: Simple and safe Practical approaches to concurrent programming are needed!

# 2 Concurrent Programming

## 2.1 Basic concepts

There are different approaches to support concurrent programming. We have *concurrent languages* such as `Java` and `Ada`, where concurrency support is directly provided by the language. Alternatively, we can have sequential languages with *libraries* that makes available for applications the operating system concurrency mechanisms (this is the case, for example, of the languages `C` and `C++`).

Address space (see figure 1):

- separated: several *processes* executing in the operating system
- shared: a process with several executing *threads*.

## 2.2 Process/thread Communication

- There are basically two models:
  - Message passing (direct communication);


– Shared memory (indirect communication).


- Communication can be *synchronous* or *asynchronous*; That is: ensuring, or not, an causality in the communication result.
- *Synchronization* required to ensure sanity in the communication.

### 2.3 New challenges

Concurrent programs introduce new challenges compared with the sequential programming.

1. *Safety*; A concurrent program is termed *safe* if it does not produce meaningless results (as a result, of course, of a concurrent execution).
2. *Liveness*. A concurrent program is termed *alive* if it is guaranteed that eventually it will achieve the desired results (regardless of the scheduling of its competitors subprograms).

Examples of these new problems are the following:

1. *race conditions* This type of safety problem – typical within low-level concurrent programming, is the result of situations where the correct use of the shared areas may depend on uncontrollable events such as process/thread scheduling. This failure can cause very serious errors in concurrent programs, since they are non-deterministic and are often difficult to replicate.
   The classic solution to prevent these errors is to define critical regions in mutual exclusion involving all uses of shared data.
2. *deadlock, starvation* The use of mutual exclusion to solve the problem of race conditions may generate a liveness problem: *deadlocks*.
   This possibility occurs whenever there is a waiting cycle to access critical regions.
   Another liveness problem - *starvation* - occurs when it is not given the possibility for a process/thread to access a shared resource by other threads or by the scheduler.

## 3 Concurrent programming in native Java

### Threads

- Native support for concurrency in Java uses threads (shared memory communication model)
- The launch of new threads is done by creating objects of type `Thread`, and invoking `start` service.
- This creation can be made in two ways:
  1. Extending `Thread` class and redefining `run` method (which will contain the thread's `main` program). In this case the default no argument constructor should be used.
  2. Implementing the `Runnable` interface and defining the `run` method. An appropriate `Runnable` object should be passed to the constructor of the `Thread` class.
- Note that beginning at Java 8, an argumentless `void` function is also considered to be a `Runnable`. Hence, lambda notation, or an appropriate method reference can be used instead of an explicit `Runnable` object.

## Thread creation (1)

```
public class MyThread extends Thread {
   public void run() {
      // the thread's program...
   }
}

...

   public void main(String[] args)  {
      MyThread t = new Mythread();
      t.start(); // thread creation and execution
      ...
   }
```

## Thread creation (2: with objects)

```
public class MyThread implements Runnable {
   public void run() {
      // the thread's program...
   }
}

...

   public void main(String[] args)  {
      Thread t = new Thread(new MyThread());
      t.start(); // thread creation and execution
      ...
   }
```

## Thread creation (2: with lambdas and method references)

```
...

   public void main(String[] args)  {
      Thread t = new Thread(() -> thread program);
      t.start(); // thread creation and execution
      ...
   }
```

```
public class C {
...

   public static void aProgram() {
      // the thread's program...
   }

   public void main(String[] args)  {
      Thread t = new Thread(C::aProgram);
      t.start(); // thread creation and execution
      ...
   }
}
```

## Thread class and Runnable interface

```
package java.lang;

public class Thread implements Runnable {
```

```
   // constructors:
   public Thread();
   public Thread(Runnable target);
   public Thread(Runnable target, String name);
   public Thread(String name);
   // methods:
   public static Thread currentThread();
   public void run();
   public void start();
   ...
}
```

```
package java.lang;

public interface Runnable {
   public void run();
}
```

## Thread termination

- A thread execution may terminate for several reasons:
  1. when its associated `run` method terminates (either normally, or as a result of an exception);
  2. by calling the method `System.exit()` (it ends not only the *thread* but also the whole program);
  3. when the method `destroy` (from the `Thread` class) is invoked;
  4. when the method `stop` (from the `Thread` class) is invoked.
- Note that the last two options *should not* be used to end the thread since the objects may be in inconsistent states (making unpredictable the future behavior of the program).

## Interrupt Threads

- The alternative to stop the execution of a thread is to implement a specific synchronization for this purpose, or use the methods: `interrupt`, `isInterrupted` or `interrupted`;
- The method `interrupt` interrupts the thread attached to the `Thread` object.

```
package java.lang;

public class Thread implements Runnable {
   ...

      // interrupt thread attached to object.
   public void interrupt();

      // thread attached to object interrupted?
   public boolean isInterrupted();

      // current thread interrupted?
   public static boolean interrupted();
}
```

## Types of Threads

- There are two types of threads in `Java`:
  - *user threads* (by default);
  - *daemon threads*.

- *Daemon threads* are terminated when there is no *user threads* are running.
- The method `setDaemon` allows the definition of the type of a thread (it must be invoked before the thread starts its execution);
- A *thread* may wait for the termination of another thread by the use of the method `join`.

### Thread Class: Termination

```
package java.lang;

public class Thread implements Runnable {
   ...
     // DEPRECATED methods:
   public void destroy();
   public void stop();
   public void stop(Throwable e);

     // usable methods:
   public boolean isAlive();
   public boolean isDaemon();
   public void join() throws InterruptedException;
   public void join(long millis)
      throws InterruptedException;
   public void join(long millis, int nanos)
      throws InterruptedException;
   public void setDaemon(boolean on);
   ...
}
```

## 3.1 Mutual exclusion

- In Java, the communication model between threads is the shared memory (within a program, all threads share the same address space);
- In order for inter-thread communication to be safe and effective a proper *synchronization* between threads is required.
- The language has direct support for this purpose through an object (or class) mutual exclusion mechanism activated by a method or block `synchronized` construct:

```
   public synchronized void f() {
      ...
   }

   public void f() {
      synchronized(this) {
         ...
      }
   }
```

# 4 Java: Memory Model

Consider the following class:

```
public class ASynch {
   private int a = 0;
   private int b = 0;

   public void set() {
     a++;
     b++;
   }
```

```
    public boolean invariant() {
        return a == b;
    }
}
```

In a `Java` sequential program this invariant is always true. However, if an `ASynch` object is shared by several threads then, as strange as it might appear, the invariant might fail (i.e. the result of function `invariant` might be false). The problem is solved, obviously, by properly synchronizing the shared object.

Trying to meet the ongoing development in computer architectures, the `Java` language defines a set of rules and minimum guarantees in the access to global memory shared by threads. This semantics is called the **memory model**.

These rules are described in three different aspects:

- **Atomicity** What instructions must have indivisible effects?
- **Visibility** Under what conditions are visible the execution effects of a thread in another thread?
- **Ordering** Under what conditions operations may appear out of order?

It should be noted that nothing prevents `Java` compilers of implementing stronger semantics of this model (for example, always ensuring a sequential alike order). So there is the possibility of incorrect programs to work in same Java virtual machines (and not in others).

## 4.1 Java: Atomicity

- Accesses and updates of attributes of any type (of primitive and non-primitive) are atomic, except for types `long` and `double`;
- Attributes declared as `volatile` are always atomic (including `long` and `double`).
- This property ensures that when a thread modifies an atomic value, others threads never observe an intermediate value (either the value it has before, or the new value).

## 4.2 Java: Visibility

Changes to attributes made by a thread (writer) are guaranteed to be visible to other threads (readers) in the following conditions:

- A writer thread releases a native lock and readers threads acquire that lock (the release of a native lock forces the writing in the global memory of all variables modified by the thread; in the other hand, the acquisition of a lock makes that all values in attributes accessible by the thread to be reread);
- Any value written to a `volatile` attribute is visible to latter reader thread (works like a kind of getter/setter `synchronized` block).
- The first time a thread reads an attribute, it will see either its initial value, or any value that has been written by another thread.
- When a thread terminates, all the modifications are written to global memory.

## 4.3 Warning on `volatile` attributes

Note that although `volatile` attributes ensure atomicity, visibility and ordering in their usage, such guaranties only apply **individually** in each of the variable operations. This means that operations as simple, such as an increment of a volatile integer variable, are not atomic, because they imply two separate operations (one to read the previous value and other to assign the new incremented value). Between these two operations there is the possibility of a race condition (probably with a very low probability of occurrence which only makes the problem even worse).

## 4.4 Java: Ordering

The rules to impose on ordering of instructions apply in two situations: inside a thread's program (*intra-thread*), and between threads (*inter-thread*):

- `intra-thread`: the executions of instructions by a thread inside a method works with the semantics as if it was sequential (meaning that reorder is allowed as long as the result is equivalent to its unordered sequential version);
- `inter-thread`: if a thread is observing attributes modified by other threads, any ordering may occur, except if those attributes are `volatile`, or inside a `synchronized` method/block.

Hence, to ensure the expected ordering semantics (as-if-sequential), it is necessary to protect shared values with `volatile` or `synchronized` methods/blocks.