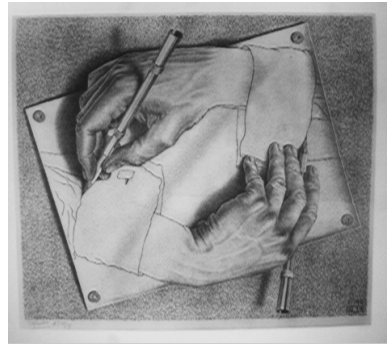
Principles of Concurrent and Distributed Programming

Emilio Tuosto

Academic Year 2025/2026

January 2026

Concurrency in Java



Processes vs. Threads

Multitasking

Many activities at once none of which "is aware" of the others (e.g., time slicing)

Processes

Running programs with their own execution environment containing basic run-time resources e.g. the processes' **address space**.

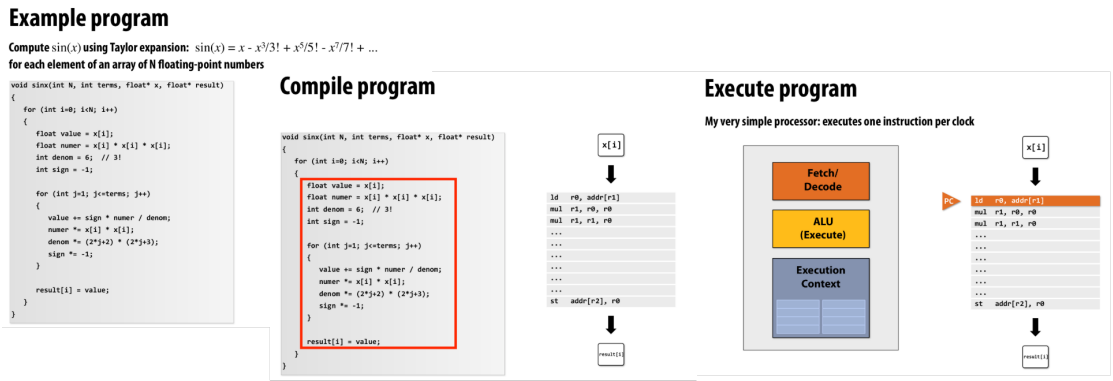
Threads

Sequential flows of control within a process (a process can consist of many **concurrent** threads)

> *Threads are also known as lightweight processes because creating a new thread requires fewer resources than creating a new process. Threads "lives" within a process and can share the process's resources (e.g., memory, files). In general multi-threaded applications have a "main" thread which can create new threads.*

Context switching

A (simplified) view of how processes interleave:



Borrowed from `https://maxnilz.com/docs/006-arch/001-cpu-basics/`

Programming with threads

From [Eck02]

Concurrent programming is like stepping into an entirely new world and learning a new programming language, or at least a **new set of language concepts**. With the appearance of thread support in most microcomputer operating systems, extensions for threads have also been appearing in programming languages or libraries. In all cases, thread programming:

- Seems **mysterious** and requires a shift in the way you think about programming
- Looks similar to thread support in other languages, so **when you understand threads, you understand a common tongue**.

[...] threads are tricky.

Concurrency and Java OO

Some documentation

`https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`

There is no general approach to concurrent programming.

Some rule of thumb

Java's "style" suggests

- to individuate active and passive objects
  - an active object is basically an object representing a thread
  - a passive object represents a resource that can be concurrently accessed by active objects
- Reason about how objects "interacts"
  - how does active objects' execution interleave?
  - how do active objects access shared resources?
- Acquire/release policy
  - in which order active objects acquire shared resources?
  - under which conditions shared resources can be invoked?
  - do active objects release all the acquired resources when they are not any longer needed?

Threads in Java

- Runnable: interface (method run() to be implemented)
- Threads: class (implements Runnable, run() is just empty)
  - Constructors
    - Thread()
    - Thread(Runnable target)
    - Thread(Runnable target, String name)
    - Thread(String name)
    - ... (see the Java thread API)
  - start, sleep, yield
  - interrupt

"When something has a Runnable interface, it simply means that it has a run() method, but there's nothing special about that –it doesn't produce any innate threading abilities, like those of a class inherited from Thread." [Eck02]

To create and run Java thread from a Runnable object:

- create the Runnable object
- use the special Thread constructors with runnable objects
- run the thread by invoking its start() method (which performs some initialisations and then calls the run() method)

Some examples

A simple scenario

Write a program that

- decides if staff is worth a promotion according to their state of service
- prints a report about the decision

Let's consider some solutions

- introducing some Java primitives for threads
- and showing how tricky concurrency can be

Some examples

A simple scenario

Write a program that

- decides if staff is worth a promotion according to their state of service
- prints a report about the decision

Let's consider some solutions

- introducing some Java primitives for threads
- and showing how tricky concurrency can be

Don't do this at home!

- PromotionConcurrent.java: a first attempt
- PromotionMoreConcurrent.java: an improved version
- IoC.java: pausing threads

Controlling threads

interrupt(): interrupts the thread on which it is invoked

yield(): Occasionally, a thread can decide to "give a hint to the thread scheduling mechanism" ([Eck02]) that it is keen to pass the control to another thread.
In Java this is done by invoking the yield() method from run.

join(): when invoked on a thread object, the invoking thread waits for the first thread to complete before proceeding (there is also a version with timeout). join() must be withing a try-catch statement because an interrupt() signal can abort the calling thread.

isAlive(): returns 'true' if the thread is running.

Mutual exclusion in Java

The mechanism that is offered by Java is method synchronisation

- Synchronised Methods can prevent thread interference and memory consistency errors
- Synchronisation based on (implicit) locks

The synchronized modifier can be used in method declarations or for determining critical sections.

- A method declared synchronized cannot be invoked while another synchronised method is executing
- (hence) If more than 2 threads try to invoke a synchronised method, only one of them actually access the object, while the other is blocked
- synchronized(obj){stm}: acquires the lock on obj, executes stm and releases the lock; stm is the critical section on the shared resource obj

Semaphores in java

```
public class Semaphore {
    private int counter = 0;
    private int threshold = 0;
    public Semaphore(int counter) { this.counter = counter; }

    public synchronized void P() {
                while(this.counter <= threshold) {
                        try {
                                this.wait();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                this.counter--;
    }

    public synchronized void V() {
                this.counter++;
                if ( this.counter - 1 == threshold )
                        // this.notifyAll();
                        this.notify();
    }
}
```

Monitors in Java

It is important to remind that waiting threads must be notified before releasing the shared object

- public final void wait() throws InterruptedException
  - The thread is suspended and it is put on the object waiting list
- public final void wait(long timeout) throws InterruptedException
  - The thread is suspended until another wakes it up or until the time is elapsed
- public final void notify()
  - Choses and Wakes up a single thread among those waiting on the object monitor.
  - Which thread is chosen depends on the implementation of the JVM
  - This method should only be called by the "owner thread", namely the one which is
    - executing a synchronized statement that synchronizes on the object
    - executing the body of a synchronized statement that synchronizes on the object
  - Throws: IllegalMonitorStateException - if the current thread does not own the object
- public final void notifyAll()
  - Like notify, but awakes all the waiting threads

Remote Method Invocation in Java

https://docs.oracle.com/en/java/javase/24/docs/specs/rmi/intro.html

RPC vs RMI

Remote Method Invocation (RMI) is the Java correspondent of RPC.

- Instead of remote procedure calls, RMI implements remote method calls (i.e., calls of methods of remote objects)
- a key difference between RPC and Java RMI is that the latter allows Java objects to communicate, while the former provides, in general, a communication middleware for programs written in different languages
- RPC can be seen as a very primitive form of message oriented middleware and is data oriented. Java RMI, on the contrary, you can communicate objects, namely data & behaviour!

Remark

Observe that Java RMI allows objects running in a JVM to invoke methods of (Java) objects running in a different JVM

Distributed Objects: some terminology

- Distributed object : an object whose methods can be remotely invoked. A distributed object is provided, or exported by the object server .
- Remote method : a public method of a distributed object.
- Object registry : is the equivalent of the RPC name server. Namely, it is used by object servers to register their services and by object clients to look up for service references.
- Client/server proxy : is the equivalent of client/server stubs in RPC. Object clients call a remote method appear direct at the programmer level. However,
  - on the client host, the client proxy interacts with the software providing runtime support for the distributed object system
  - the runtime support transmits the actual call to the remote host (it also marshals the parameter to be transmitted)
  - similarly, on the object server side, the runtime support for the distributed object system handles the incoming messages (and their unmarshalling), and forwards the call to the server proxy

Java RMI: the first step

In Java:

- remote objects are those objects extending the java.rmi.Remote remote interface . Basically, interfaces plays the role of ID, hence the IDL of Java RMI is java.rmi.Remote
- the object server
  - implements a remote interface
  - generates stub and skeleton
  - register a distributed object implementing the interface
- An object client accesses the object by invoking the remote methods associated with the objects using syntax provided for remote method invocations

Remark

Within RMI, remote objects are treated differently from non-remote objects. For instance, what RMI actually passes when a remote object reference r_obj is sent to a remote object is a remote stub for r_obj. The stub acts as the local proxy for r_obj so that the caller is unaffected and calls r_obj via its stub.

Java RMI: the second step

Applications relying on distributed object must:

- Locate remote objects
  - by passing remote object references or
  - by using the object registry
- Communicate with remote objects
- Load class bytecodes for objects passed around: since RMI allows a caller to pass objects within calls to remote methods, RMI yields the necessary mechanisms for loading an object's code and for transmitting its data.

Remark

One of the central features of RMI is the possibility of dinamically downloading bytecodes of the class of an object when it is not defined in the caller's JVM. Basically, the types and the behavior of an object can be transmitted to possibly remote JVMs. RMI guarantees that the behavior of objects remains unchanged when they are sent to another JVM and allows new types to be introduced into a remote virtual machine, so that an application can be dynamically extended .

Creating Distributed Applications Using RMI

Using RMI to develop a distributed application requires you to follow these general steps:

1. Design and implement the distributed application components
2. Compile sources and generate stubs
3. Make classes network accessible: In this step you make everything–the class files associated with the remote interfaces, stubs, and other classes that need to be downloaded to clients–accessible via a Web server.
4. Start the application: Starting the application means to run:
   1. the RMI remote object registry
   2. the server
   3. the client

1. Design and implement the distributed application components

First, give an initial architecture for your application (this might require some revision at a later stage) and determine which components are local objects and remote objects .
This phase consists of:

- remote interfaces definition : this specifies the remote methods When designing remote interfaces you have to determine any local objects that will be used as parameters and return values for these methods
- remote objects implementation : generally, remote objects have to implement several remote interfaces (of course, the remote object class may implements other non-remote interfaces and define methods available only locally). Any local classes used in remote method invocations (as parameters or return values) must be implemented.
- clients implementation : clients invoking remote objects can be implemented at any time after the definition of remote interfaces or after deployment of remote objects.

2. Compile sources and generate stubs

This phase has two steps:

- use javac to compile the server classes (those implementing remote interfaces) and the client classes
- use rmic compiler in order to create stubs for remote objects.

Remark

The Java rmic compiler generates the stubs, namely, the programmer does not have to program client and server proxies and low lever programming detail.

Java remote interface

- In a remote interface each method signature must throw RemoteException Other than this, a remote interface has the same syntax as any other Java interface.
- RemoteException exception is raised if errors occur when processing remote method call. The exception is must be caught by the caller.
- RemoteException can be caused
  - by exceptions that may occur during communications (e.g., access or connection failures)
  - by problems in remote method invocations (e.g., errors resulting from object, stub, or skeleton not being found)

An example:

```
import java.rmi.*;

public interface ARemoteInterface extends Remote {
    String aRemoteMethod1( ... ) throws RemoteException;
    int aRemoteMethod2( ... ) throws RemoteException;
}
```

An example: the compute engine

The compute engine is a protocol to execute tasks on a remote engine. This protocol is based on interfaces supported by the compute engine and by the objects that are submitted to the compute engine.
The remotely accessible part is the compute engine itself, whose remote interface has a single method:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
/*
 * The 2 lines above can be replaced by
 * import java.rmi.*;
 */
public interface Compute extends Remote {
    public Object executeTask(Task t)
        throws RemoteException;
}
```

By extending java.rmi.Remote, the interface Compute allows its method to be called from any JVM. Any object implementing Compute becomes a remote object.
Notice that executeTask

- takes a Task
- can return any Object
- throws RemoteException

An example: the compute engine (2)

An interface for Task objects must be defined.

```
import java.io.Serializable;

public interface Task extends Serializable {

    public Object execute();

}
```

Different kinds of tasks can be run by a Compute object provide that they implementat Task. It is possible to add further methods (or data) needed for the computation of the task.

Exercise

execute is not required to throw RemoteException. Why?

Remark

The Task interface extends the java.io.Serializable interface to let the RMI middleware serialise objects so that they can be transported from a JVM to another.
Implementing Serializable marks the class as being capable of conversion into a self-describing byte stream that can be used to reconstruct an exact copy of the serialized object when the object is read back from the stream.

This implies that local objects are passed by-value while remote objects are passed by-reference .

[Eck02] Bruce Eckel. *Thinking in Java, 4rd Edition*.
Prentice-Hall, December 2002. Chapter 13. The beta version of the 3rd edition is available at
http://www.javaclue.org/pub/java/ebooks/tij/tij-3rd-ed.pdf.