BABEŞ-BOLYAI UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

# Java 8 Concurrency

*Authors:*
Sergiu BREBAN, Sisteme Distribuite

January 16, 2017

# Contents

**1 General Java Concurrency** **1**
1.1 Basic concurrency concepts . . . . . 1
1.1.1 Syncronization . . . . . 1
1.1.2 Concurrent applications possible problems . . . . . 2
1.2 Traditional Java Concurrency Model . . . . . 2
1.2.1 The Java Thread Model . . . . . 2
1.2.2 The Main Thread . . . . . 3
1.2.3 Creating a thread . . . . . 3
1.2.4 Thread priorities . . . . . 4
1.3 Syncronization . . . . . 4
1.3.1 Using syncronized Methods . . . . . 4
1.3.2 Using syncronized Statement . . . . . 5
1.3.3 Using Syncronization Objects . . . . . 5
1.4 Using an Executor . . . . . 5
1.5 Parallel Programming via the Fork/Join Framework . . . . . 6

**2 Java 8: Streams and Lambda expressions** **7**
2.1 Lambda Expression . . . . . 7
2.1.1 Where and how to use lambdas . . . . . 8
2.2 Parallel streams . . . . . 8

**Bibliography** **11**

# List of Figures

1.1 Thread class methods . . . . . 2

# Chapter 1

# General Java Concurrency

## 1.1 Basic concurrency concepts

Concurrency and parallelism are similar concepts. Concurrency generally refers to a situation when you have more than one task in a single processor with a single core, and the task scheduler from the operating system level quickly switches between tasks, such they appear as running simultaneosly. Parallelism is when you have more than one task that run simultaneously on a different processor, core inside a processor or even a different computer. Another definition shows that parallelism refers to having different instances of the same task running at the same time over different parts of a data set. [1]

### 1.1.1 Syncronization

Syncronization is defined as the coordination of two or more tasks to get the desired results. It is of two kinds:

- Control syncronization, when on task depends on another task result
- Data access syncronization, when multiple tasks have access to a shared variable and only one can access it at any given time

A **critical section** is a piece of code that is executed by a single task at any time, because it access a shared resource. Mutual exclusion is a mechanism used to make this possible.

From a theoretical point of view, the popular mechanism used to get synchronization in a concurrent system are:

- Semaphore
- Monitor

A piece of code is thread safe if it can be used in concurrent applications without problems, by using a syncronization mechanism, a nonblocking compare and swap primitive or immutable data. An **immutable object** is thread-save, because the value of its attributes cannot be modified after initialisation, so you have to create a new one if you want to change it.

An **atomic operation** is an operation that appers as occcuring simultaneosly to the other tasks, and is implemented using syncronization mechanisms, and an **atomic variable** is a variable whose value is got and set using atomic operations.

The concurent tasks have two methods to communicate with each other. One is **shared memory**, and is used when the tasks read and write data from the same memory area, and **message passing**, used when tasks are running on different computers and nedd to comunicate with each other.

### 1.1.2 Concurrent applications possible problems

Some of the problems that appear in concurrent applications, caused by the wrong use of syncronization mechanism, are [1]:

- **Data race (race condition)**, when multiple tasks try to write a shared variable without using any syncronization mechanisms
- **Deadlock** occurs when two or more tasks wait for the others to free a shared resource, so they are blocked indefinitely and none will get the resource
- A **livelock** occurs when two tasks are always changing their states due to the actions of the other, they are in a loop, unable to continue
- **Resource starvation** appears when a task never gets the resource it needs to continue, and **fairness** is the solution for that problem
- **Priority inversion** appears when a resource is holden by a low-priority task and a high-priority task needing it cannot continue its execution

## 1.2 Traditional Java Concurrency Model

### 1.2.1 The Java Thread Model

Java uses threads to enable its entire environment to be anynchronous, reducing inefficiency by preventing the waste of CPU cycles. This multithreading system is build using the **Thread** class with its methods and the **Runnable** interface. A thread of execution is encapsulated in a **Thread** instance. You have to extend the **Thread** class or to implement the **Runnable** interface to create a new thread. Some of the methods defined in the **Thread** class to help you manage threads are presented in 1.1 [2]

| Method | Meaning |
|---|---|
| getName | Obtain a thread's name. |
| getPriority | Obtain a thread's priority. |
| isAlive | Determine if a thread is still running. |
| join | Wait for a thread to terminate. |
| run | Entry point for the thread. |
| sleep | Suspend a thread for a period of time. |
| start | Start a thread by calling its run method. |

FIGURE 1.1: Thread class methods

### 1.2.2 The Main Thread

A thread begins running when a Java program starts, and it is usually called the main thread. It is used to spawn other child threads and it is the last thread finishing execution and performs some shutdown actions.

### 1.2.3 Creating a thread

**Implementing Runnable** To implement Runnable, a class needs to implement only the **run()** method. Inside this method you define the code that constitues the new thread. The thread will end when run() returns. You have to instantiate a Thread object from within the class that implements Runnable.

```
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

**Extending Thread** Another way to create a thread is to create a class that extends Thread and instantiate it. The entry point of the new thread is the **run()** method, which has to be overridden. To begin execution of the thread, the new class has to call the start() method.

```
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
```

```
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

**isAlive() and join()** You will often want the main thread to finish last. This can be accomplished by calling sleep(), but the most useful methods are isAlive(), whic returns true if the thread is still alive, or false otherwise, and join(), which waits until the thread on which is called terminates.

```
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
ob1.t.join();
ob2.t.join();
ob3.t.join();
```

### 1.2.4 Thread priorities

The thread scheduler uses thread priorities to decide when a thread should be allowed to run. To set the priority of a thread, we can use the **setPriority()** method from the Thread class. The values must be in the range `MIN_PRIORITY and MAX_PRIORITY`, currently these values being 1 and 10. The current priority of a thread can be obtained using the **getPriority()** method.

## 1.3 Syncronization

### 1.3.1 Using syncronized Methods

The easiest syncronization method in Java is to use the monitor associated by each object. To enter this monitor, you have to call all method associated with the **synchronized** keyword. All the other threads that want to call the syncronized method while other thread is inside it have to wait for it to finish.

### 1.3.2 Using syncronized Statement

To access an object of a class that does not use syncronized methods, you just have to put the calls to that methods inside a syncronized block. The general form of a syncronized block is:

```
syncronized(objRef) {
   //statements to be syncronized
}
```

### 1.3.3 Using Syncronization Objects

To enable handling of several difficult syncronization situations easily, the following syncronization objects are supported: **Semaphore, CountDownLatch, CyclicBarrier, Exchanger and Phaser**. All these classes are located in the java.util.concurrent package, which defines some core features to support advanced syncronization and interthread communication methods.

**Semaphore**

**CountDownLatch**

**CyclicBarrier**

**Exchanger**

**Phaser**

## 1.4 Using an Executor

The executor framework is a mechanism that allows you to separate thread creation and management for the implementation of concurrent tasks. You don't have to worry about the creation and management of threads, only about creating tasks and sending them to the executor. The main classes involved in this framework are [1]:

- The Executor and ExecutorService interface
- ThreadPoolExecutor
- ScheduledThreadPoolExecutor
- Executors
- The Callable interface
- The Future interface

## 1.5 Parallel Programming via the Fork/Join Framework

The fork/join framework was designed to recursively split a parallelizable task into smaller tasks and then combine the results of each subtask to produce the overall result. It's an implementation of the ExecutorService interface, which distributes those subtasks to worker threads in a thread pool, called ForkJoinPool. To submit tasks to this pool, you have to create a subclass of RecursiveTask<R>, where R is the type of the result produced by the parallelized task (and each of its subtasks) or of RecursiveAction if the task returns no result (it could be updating other nonlocal structures, though). To define RecursiveTasks you need only implement its single abstract method, compute:

```
protected abstract R compute();
```

This method defines both the logic of splitting the task at hand into subtasks and the algorithm to produce the result of a single subtask when it's no longer possible or convenient to further divide it. [3]

# Chapter 2

# Java 8: Streams and Lambda expressions

Streams and lamba expressions are the most important new features introduced in Java 8 version. Stream has been added as a method in the Collection interface and other data sources. They allow processing the elements of a data structure to generate new structures and filter the data. They can be used to implement algorithms using the map and reduce technique.

Parallel streams are a special kind of streams which operates in a parallel way. The elements involved in the use of parallel streams are:

- The Stream interface: defines the operations that can be performed on a stream.
- Optional: a container object
- Collectors: implements the reduction operations that are used in a stream sequence of operations.
- Lambda expressions: most stream methods accept a lambda expression as a parameter, allow more compact operations implementaions.

## 2.1 Lambda Expression

A lambda expression can be understood as a concise representation of an anonymous function that can be passed around: it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown.

- Anonymous - it doesn't have an explicit name like a method would normally have.
- Function - a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- Passed around - a lambda expression can be passed as argument to a method or stored in a variable.
- Concise - you don't need to write a lot of boilerplate like you do for anonymous classes.

Lambdas technically don't let you do anything that you couldn't do prior to Java 8. But you no longer have to write clumsy code using anonymous classes to benefit from behavior parameterization. The net result is that your code will be clearer and more flexible. For example, using a lambda expression you can create a custom Comparator object in a more concise way:

Without lambda expression:

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
};
```

With lambda expression:

```
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

### 2.1.1 Where and how to use lambdas

**Functional interface** You can use a lambda expression in the context of a functional interface. A functional interface is an interface that specifies exactly one abstract method. You already know several other functional interfaces in the Java API such as Comparator and Runnable. Interfaces can now also have default methods (that is, a method with a body that provides some default implementation for a method in case it isn't implemented by a class). An interface is still a functional interface if it has many default methods as long as it specifies only one abstract method. Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and treat the whole expression as an instance of a functional interface (more technically speaking, an instance of a concrete implementation of the functional interface). You can achieve the same thing with an anonymous inner class, although it's clumsier: you provide an implementation and instantiate it directly inline.

**Function descriptor** The signature of the abstract method of the functional interface essentially describes the signature of the lambda expression. We call this abstract method a function descriptor. For example, the Runnable interface can be viewed as the signature of a function that accepts nothing and returns nothing (void) because it has only one abstract method called run, which accepts nothing and returns nothing (void). [3]

## 2.2 Parallel streams

Stream interface allows you to process its elements in parallel in a very convenient way: it's possible to turn a collection into a parallel stream by invoking the method parallelStream on the collection source. A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread. Thus,

you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.

Let's suppose you need to write a method accepting a number n as argument and returning the sum of all the numbers from 1 to the given argument. A straightforward approach is to generate an infinite stream of numbers, limiting it to the passed number, and then reduce the resulting stream with a BinaryOperator that just sums two numbers, as follows:

```
public static long sequentialSum(long n) {
   return Stream.iterate(1L, i -> i + 1)
                .limit(n)
                .reduce(0L, Long::sum);
}
```

In more traditional Java terms, this code is equivalent to its iterative counterpart:

```
public static long iterativeSum(long n) {
    long result = 0;
    for (long i = 1L; i <= n; i++) {
        result += i;
    }
    return result;
}
```

This operation seems to be a good candidate to leverage parallelization, especially for large values of n.

You can make the former functional reduction process (that is, summing) run in parallel by turning the stream into a parallel one; call the method parallel on the sequential stream:

```
public static long parallelSum(long n) {
   return Stream.iterate(1L, i -> i + 1)
                .limit(n)
                .parallel()
                .reduce(0L, Long::sum);
}
```

Note that, in reality, calling the method parallel on a sequential stream doesn't imply any concrete transformation on the stream itself. Internally, a boolean flag is set to signal that you want to run in parallel all the operations that follow the invocation to parallel. Similarly, you can turn a parallel stream into a sequential one by just invoking the method sequential on it.

Parallel streams internally use the default ForkJoinPool, which by default has as many threads as you have processors, as returned by Runtime. getRuntime(). availableProcessors(). But you can change the size of this pool using the system property java. util. concurrent. ForkJoinPool. common. parallelism, as in the following example:

```
   System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism",
       "12");
```

This is a global setting, so it will affect all the parallel streams in your code. Conversely, it currently isn't possible to specify this value for a single parallel stream. In general, having the size of the ForkJoinPool equal to the number of processors on your machine is a meaningful default.

# Bibliography

[1] Javier Fernandez Gonzalez. *Mastering concurrency programming with Java 8: master the principles and techniques of multithreaded programming with the Java 8 concurrency API*. Community experience distilled. Birmingham: Packt Publ., 2016.

[2] Herbert Schildt and Danny Coward. *Java: The Complete Reference, Ninth Edition*. McGraw-Hill Education, 2014.

[3] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and functional-style programming*. Manning Publications Co., 2014.