Mastering Concurrency Programming with Java 8

Concurrency programming allows several large tasks to be divided into smaller sub-tasks, which are further processed as individual tasks that run in parallel. All the sub-tasks are combined once the required results are achieved; they are then merged to get the final output. Java includes a comprehensive API with a lot of ready-to-use components to implement powerful concurrency applications in an easy way, but with high flexibility to adapt these components to your needs.

The book starts with a description of the design principles of concurrent applications and how to parallelize a sequential algorithm. Next, you will learn to use the most important components of the Java 8 concurrency API: the executor framework, the Phaser class, and the Fork/Join framework. Towards the end, we will cover the new additions to the Java 8 API, the Map and Reduce model, and the Map and Collect model. The book will also teach you about the data structures and synchronization utilities to avoid data-race conditions and other critical problems. Finally, the book ends with a detailed description of the tools and techniques that you can use to test a Java concurrent application.

Who this book is written for

If you are a competent Java developer with a good understanding of concurrency, but have no knowledge of how to effectively implement concurrent programs or use streams to make processes more efficient, then this book is for you.

What you will learn from this book

- Design concurrent applications by converting a sequential algorithm into a concurrent one
- Discover how to avoid all the possible problems you can get in concurrent algorithms
- Use the executor framework to manage concurrent tasks without creating threads
- Extend and modify executors to adapt their behavior to your needs
- Solve problems using the divide and conquer technique and the Fork/Join framework
- Process massive data sets with parallel streams and MapReduce implementation
- Control data-race conditions using concurrent data structures and synchronization mechanisms
- Test and monitor concurrent applications

Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 *The First Step – Concurrency Design Principles*
- A synopsis of the book’s content
- More information on *Mastering Concurrency Programming with Java 8*
Javier Fernández González is a software architect with almost 15 years' experience with Java technologies. He has worked as a teacher, researcher, programmer, analyst, writer, and now as an architect in all types of projects related to Java, especially J2EE. As a teacher, he has taught over 1,000 hours of training in basic Java, J2EE, and Struts framework. As a researcher, he has worked in the field of information retrieval, developing applications in order to process large amounts of data in Java and has been a part of several journal articles and conference presentations as a coauthor. In recent years, he has worked on developing J2EE web applications for various clients from different sectors (public administration, insurance, healthcare, transportation, and so on). Currently, he is working as a software architect at Capgemini, which includes developing and maintaining applications for an insurance company. Also, he is the author of the book Java 7 Concurrency Cookbook, Packt Publishing.
Nowadays, computer systems (and other related systems, such as tablets or smartphones) allow you to do several tasks simultaneously. This can be possible because they have concurrent operating systems that control several tasks at the same time. You can also have one application that executes several tasks (read a file, show a message, or read data over a network) if you work with the concurrency API of your favorite programming language. Java includes a very powerful concurrency API that allows you to implement any kind of concurrency application with little effort. This API increases the features provided to programmers in every version. Now, in Java 8, it has included the stream API and new methods and classes to facilitate the implementation of concurrent applications. This book covers the most important elements of the Java concurrency API, showing you how to use them in real-world applications. These elements are as follows:

- The executor framework, to control the execution of lots of task
- The Phaser class, to execute tasks that can be divided into phases
- The Fork/Join framework, to execute the tasks that solve a problem using the divide and conquer technique
- The stream API, to process big sources of data
- Concurrent data structures, to store the data in concurrent applications
- Synchronization mechanisms, to organize concurrent tasks

However, it includes much more: a methodology to design concurrency applications, design patterns, tips and tricks to implement good concurrency applications, and tools and techniques to test concurrency applications.
What this book covers

Chapter 1, The First Step – Concurrency Design Principles, will teach you the design principles of concurrency applications. They will also learn the possible problems of concurrency applications and a methodology to design them followed by some design patterns, tips, and tricks.

Chapter 2, Managing Lots of Threads – Executors, will teach you the basic principles of the executor framework. This framework allows you to work with lots of threads without creating or managing them. You will implement the k-nearest neighbors algorithm and a basic client/server application.

Chapter 3, Getting the Maximum from Executors, will teach you some advanced characteristics of executors, including cancelation and scheduling of tasks to execute a task after a delay or every certain period of time. You will implement an advanced client/server application and a news reader.

Chapter 4, Getting Data from the Tasks – The Callable and Future Interfaces, will teach you how to work in an executor with tasks that return a result using the Callable and Future interfaces. You will implement a best-matching algorithm and an application to build an inverted index.

Chapter 5, Running Tasks Divided into Phases – The Phaser class, will teach you how to use the Phaser class to execute tasks that can be divided into phases in a concurrent way. You will implement a keyword extraction algorithm and a genetic algorithm.

Chapter 6, Optimizing Divide and Conquer Solutions – The Fork/Join Framework, will teach you how to use a special kind of executor optimized by those problems that can be resolved using the divide and conquer technique: the Fork/Join framework and its work-stealing algorithm. You will implement the k-means clustering algorithm, a data filtering algorithm, and the merge-sort algorithm.

Chapter 7, Processing Massive Datasets with Parallel Streams – The Map and Reduce Model, will teach you how to work with streams to process big datasets. In this chapter, you will learn how to implement map and reduce applications using the stream API and much more functions of streams. You will implement a numerical summarization algorithm and an information retrieval search tool.

Chapter 8, Processing Massive Datasets with Parallel Streams – The Map and Collect Model, will teach you how to use the collect() method of the stream API to perform a mutable reduction of a stream of data into a different data structure, including the predefined collectors defined in the Collectors class. You will implement a tool to search data without indexing, a recommendation system, and an algorithm to calculate the list of common contacts of two persons in a social network.
Chapter 9, *Diving into Concurrent Data Structures and Synchronization Utilities*, will teach you how to work with the most important concurrent data structures (data structures that can be used in concurrent applications without causing data race conditions) and all the synchronization mechanisms included in the Java concurrency API to organize the execution of tasks.

Chapter 10, *Integration of Fragments and Implementation of Alternatives*, will teach you how to implement a big application made by fragments of concurrent applications with their own concurrency techniques using shared memory or message passing. You will also learn different implementation alternatives to the examples presented in the book.

Chapter 11, *Testing and Monitoring Concurrent Applications*, teaches you how to obtain information about the status of some of the Java concurrency API elements (thread, lock, executor, and so on). You will also learn how to monitor a concurrent application using the Java VisualVM application and how to test concurrent applications with the MultithreadedTC library and the Java Pathfinder application.
Users of computer systems are always looking for better performance for their systems. They want to get higher quality videos, better video games, and faster network speed. Some years ago, processors gave better performance to users by increasing their speed. But now, processors don't increase their speed. Instead of this, they add more cores so that the operating system can execute more than one task at a time. This is named concurrency. Concurrent programming includes all the tools and techniques to have multiple tasks or processes running at the same time in a computer, communicating and synchronizing between them without data loss or inconsistency. In this chapter, we will cover the following topics:

- Basic concurrency concepts
- Possible problems in concurrent applications
- A methodology to design concurrent algorithms
- The Java concurrency API
- The Java memory model
- Concurrency design patterns
- Tips and tricks to design concurrency algorithms

**Basic concurrency concepts**
First of all, let's present the basic concepts of concurrency. You must understand these concepts to follow the rest of the book.
Concurrency and parallelism are very similar concepts. Different authors give different definitions to these concepts. The most accepted definition talks about concurrency when you have more than one task in a single processor with a single core and the operating system's task scheduler quickly switches from one task to another, so it seems that all the tasks run simultaneously. The same definition talks about parallelism when you have more than one task that run simultaneously at the same time, in a different computer, processor, or core inside a processor.

Another definition talks about concurrency when you have more than one task (different tasks) running simultaneously on your system. One more definition discusses parallelism when you have different instances of the same task running simultaneously over different parts of a dataset.

The last definition that we include talks about parallelism when you have more than one task that runs simultaneously in your system and talks about concurrency to explain the different techniques and mechanisms programmers have to synchronize with the tasks and their access to shared resources.

As you can see, both concepts are very similar and this similarity has increased with the development of multicore processors.

Synchronization
In concurrency, we can define synchronization as the coordination of two or more tasks to get the desired results. We have two kinds of synchronization:

- Control synchronization: When, for example, one task depends on the end of another task, the second task can't start before the first has finished
- Data access synchronization: When two or more tasks have access to a shared variable and only one of the tasks can access the variable at any given time

A concept closely related to synchronization is critical section. A critical section is a piece of code that can be only executed by a task at any given time because of its access to a shared resource. Mutual exclusion is the mechanism used to guarantee this requirement and can be implemented by different ways.
Keep in mind that synchronization helps you avoid some errors you can have with concurrent tasks (they will be described later in this chapter), but it introduces some overhead to your algorithm. You have to calculate very carefully the number of tasks, which can be performed independently without intercommunication in your parallel algorithm. It's the granularity of your concurrent algorithm. If you have a coarse-grained granularity (big tasks with low intercommunication), the overhead due to synchronization will be low. However, maybe you won't benefit all the cores of your system. If you have a fine-grained granularity (small tasks with high intercommunication), the overhead due to synchronization will be high and maybe the throughput of your algorithm won't be good.

There are different mechanisms to get synchronization in a concurrent system. The most popular mechanisms from a theoretical point of view are:

- **Semaphore**: A semaphore is a mechanism that can be used to control the access to one or more units of a resource. It has a variable that stores the number of resources that can be used and two atomic operations to manage the value of the variable. A mutex (short for mutual exclusion) is a special kind of semaphore that can take only two values (resource is free and resource is busy), and only the process that sets the mutex to busy can release it.

- **Monitor**: A monitor is a mechanism to get mutual exclusion over a shared resource. It has a mutex, a condition variable, and two operations to wait for the condition and to signal the condition. Once you signal the condition, only one of the tasks that are waiting for it continues with its execution.

The last concept related to synchronization you're going to learn in this chapter is thread safety. A piece of code (or a method or an object) is thread-safe if all the users of shared data are protected by synchronization mechanisms, a nonblocking compare-and-swap (CAS) primitive or data is immutable, so you can use that code in a concurrent application without any problem.

**Immutable object**

An immutable object is an object with a very special characteristic. You can't modify its visible state (the value of its attributes) after its initialization. If you want to modify an immutable object, you have to create a new one.

Its main advantage is that it is thread-safe. You can use it in concurrent applications without any problem.

An example of an immutable object is the String class in Java. When you assign a new value to a String object, you are creating a new string.
Atomic operations and variables

An atomic operation is a kind of operation that appears to occur instantaneously to the rest of the tasks of the program. In a concurrent application, you can implement an atomic operation with a critical section to the whole operation using a synchronization mechanism.

An atomic variable is a kind of variable with atomic operations to set and get its value. You can implement an atomic variable using a synchronization mechanism or in a lock-free manner using CAS, which doesn't need any synchronization.

Shared memory versus message passing

Tasks can use two different methods to communicate with each other. The first one is shared memory, and normally it is used when the tasks are running in the same computer. The tasks use the same memory area where they write and read values. To avoid problems, the access to this shared memory has to be in a critical section protected by a synchronization mechanism.

The other synchronization mechanism is message passing and normally is used when the tasks are running in different computers. When a task needs to communicate with another, it sends a message that follows a predefined protocol. This communication can be synchronous if the sender is blocked waiting for a response or asynchronous if the sender continues with their execution after sending the message.

Possible problems in concurrent applications

Programming a concurrent application is not an easy job. If you incorrectly use the synchronization mechanisms, you can have different problems with the tasks in your application. In this section, we describe some of these problems.

Data race

You can have a data race (also named race condition) in your application when you have two or more tasks writing a shared variable outside a critical section — that's to say, without using any synchronization mechanisms.
Under these circumstances, the final result of your application may depend on the order of execution of the tasks. Look at the following example:

```java
package com.packt.java.concurrency;

public class Account {
    private float balance;
    public void modify (float difference) {
        float value=this.balance;
        this.balance=value+difference;
    }
}
```

Imagine that two different tasks execute the `modify()` method in the same `Account` object. Depending on the order of execution of the sentences in the tasks, the final result can vary. Suppose that the initial balance is 1000 and the two tasks call the `modify()` method with 1000 as a parameter. The final result should be 3000, but if both tasks execute the first sentence at the same time and then the second sentence at the same time, the final result will be 2000. As you can see, the `modify()` method is not atomic and the `Account` class is not thread-safe.

**Deadlock**

There is a **deadlock** in your concurrent application when there are two or more tasks waiting for a shared resource that must be free from the other, so none of them will get the resources they need and will be blocked indefinitely. It happens when four conditions happen simultaneously in the system. They are **Coffman's conditions**, which are as follows:

- **Mutual exclusion**: The resources involved in the deadlock must be nonshareable. Only one task can use the resource at a time.
- **Hold and wait condition**: A task has the mutual exclusion for a resource and it's requesting the mutual exclusion for another resource. While it's waiting, it doesn't release any resources.
- **No pre-emption**: The resources can only be released by the tasks that hold them.
- **Circular wait**: There is a circular waiting where Task 1 is waiting for a resource that is being held by Task 2, which is waiting for a resource being held by Task 3, and so on until we have Task n that is waiting for a resource being held by Task 1.
There exist some mechanisms that you can use to avoid deadlocks:

- **Ignore them**: This is the most commonly used mechanism. You suppose that a deadlock will never occur on your system, and if it occurs, you can see the consequences of stopping your application and having to re-execute it.

- **Detection**: The system has a special task that analyzes the state of the system to detect if a deadlock has occurred. If it detects a deadlock, it can take action to remedy the problem. For example, finishing one task or forcing the liberation of a resource.

- **Prevention**: If you want to prevent deadlocks in your system, you have to prevent one or more of Coffman’s conditions.

- **Avoidance**: Deadlocks can be avoided if you have information about the resources that are used by a task before it begins its execution. When a task wants to start its execution, you can analyze the resources that are free in the system and the resources that the task needs to decide that it can start its execution or not.

**Livelock**

A **livelock** occurs when you have two tasks in your systems that are always changing their states due to the actions of the other. Consequently, they are in a loop of state changes and unable to continue.

For example, you have two tasks—Task 1 and Task 2—and both need two resources: Resource 1 and Resource 2. Suppose that Task 1 has a lock on Resource 1, and Task 2 has a lock on Resource 2. As they are unable to gain access to the resource they need, they free their resources and begin the cycle again. This situation can continue indefinitely, so the tasks will never end their execution.

**Resource starvation**

**Resource starvation** occurs when you have a task in your system that never gets a resource that it needs to continue with its execution. When there is more than one task waiting for a resource and the resource is released, the system has to choose the next task that can use it. If your system has not got a good algorithm, it can have threads that are waiting for a long time for the resource.
Fairness is the solution to this problem. All the tasks that are waiting for a resource must have the resource in a given period of time. An option is to implement an algorithm that takes into account the time that a task has been waiting for a resource when it chooses the next task that will hold a resource. However, fair implementation of locks requires additional overhead, which may lower your program throughput.

Priority inversion
Priority inversion occurs when a low-priority task holds a resource that is needed by a high-priority task, so the low-priority task finishes its execution before the high-priority task.

A methodology to design concurrent algorithms
In this section, we’re going to propose a five-step methodology to get a concurrent version of a sequential algorithm. It’s based on the one presented by Intel in their Threading Methodology: Principles and Practices document.

The starting point – a sequential version of the algorithm
Our starting point to implement a concurrent algorithm will be a sequential version of it. Of course, we can design a concurrent algorithm from scratch, but I think that a sequential version of the algorithm will give us two advantages:

- We can use the sequential algorithm to test if our concurrent algorithm generates correct results. Both algorithms must generate the same output when they receive the same input, so we can detect some problems in the concurrent version, such as data races or similar conditions.
- We can measure the throughput of both algorithms to see if the use of concurrency gives us a real improvement in the response time or in the amount of data the algorithm can process in a time.
Step 1 – analysis
In this step, we are going to analyze the sequential version of the algorithm to look for the parts of its code that can be executed in a parallel way. We should pay special attention to those parts that are executed most of the time or that execute more code because, by implementing a concurrent version of those parts, we're going to get a greater performance improvement.

Good candidates for this process are loops where one step is independent of the other steps or portions of code that are independent of other parts of the code (for example, an algorithm to initialize an application that opens the connections with the database, loads the configuration files, initialize some objects. All the previous tasks are independent of each other).

Step 2 – design
Once you know what parts of the code you are going to parallelize, you have to decide how to do that parallelization.

The changes in the code will affect two main parts of the application:

- The structure of the code
- The organization of the data structures

You can take two different approaches to accomplish this task:

- **Task decomposition**: You do task decomposition when you split the code in two or more independent tasks that can be executed at once. Maybe some of these tasks have to be executed in a given order or have to wait at the same point. You must use synchronization mechanisms to get this behavior.

- **Data decomposition**: You do data decomposition when you have multiple instances of the same task that work with a subset of the dataset. This dataset will be a shared resource, so if the tasks need to modify the data you have to protect access to it by implementing a critical section.
Another important point to keep in mind is the granularity of your solution. The objective of implementing a parallel version of an algorithm is to achieve improved performance, so you should use all the available processors or cores. On the other hand, when you use a synchronization mechanism, you introduce some extra instructions that must be executed. If you split the algorithm into a lot of small tasks (fine-grained granularity), the extra code introduced by the synchronization can provoke performance degradation. If you split the algorithm into fewer tasks than cores (coarse-grained granularity), you are not taking advantage of all the resources. Also, you must take into account the work every thread must do, especially if you implement a fine-grained granularity. If you have a task longer than the rest, that task will determine the execution time of the application. You have to find the equilibrium between these two points.

**Step 3 – implementation**

The next step is to implement the parallel algorithm using a programming language and, if it's necessary, a thread library. In the examples of this book, you are going to use Java to implement all the algorithms.

**Step 4 – testing**

After finishing the implementation, you have to test the parallel algorithm. If you have a sequential version of the algorithm, you can compare the results of both algorithms to verify that your parallel implementation is correct.

Testing and debugging a parallel implementation are difficult tasks because the order of execution of the different tasks of the application is not guaranteed. In *Chapter 11, Testing and Monitoring Concurrent Applications*, you will learn tips, tricks, and tools to do these tasks efficiently.

**Step 5 – tuning**

The last step is to compare the throughput of the parallel and the sequential algorithms. If the results are not as expected, you must review the algorithm, looking for the cause of the bad performance of the parallel algorithm.

You can also test different parameters of the algorithm (for example, granularity or number of tasks) to find the best configuration.
There are different metrics to measure the possible performance improvement you can obtain parallelizing an algorithm. The three most popular metrics are:

- **Speedup**: This is a metric for relative performance improvement between the parallel and the sequential versions of the algorithm:

\[
Speedup = \frac{T_{\text{sequential}}}{T_{\text{concurrent}}}
\]

Here, \( T_{\text{sequential}} \) is the execution time of the sequential version of the algorithm and \( T_{\text{concurrent}} \) is the execution time of the parallel version.

- **Amdahl’s law**: This is used to calculate the maximum expected improvement obtained with the parallelization of an algorithm:

\[
Speedup \leq \frac{1}{(1-P) + \frac{P}{N}}
\]

Here, \( P \) is the percentage of code that can be parallelized and \( N \) is the number of cores of the computer where you’re going to execute the algorithm.

For example, if you can parallelize 75% of the code and you have four cores, the maximum speedup will be given by the following formula:

\[
Speedup \leq \frac{1}{(1-0.75) + \frac{0.75}{4}} \leq \frac{1}{0.44} \leq 2.29
\]

- **Gustafson-Barsis’ law**: Amdahl’s law has a limitation. It supposes that you have the same input dataset when you increase the number of cores, but normally, when you have more cores, you want to process more data. Gustafson law proposes that when you have more cores available, bigger problems can be solved in the same time using the following formula:

\[
Speedup = N - (1 - P) \times (N - 1)
\]

Here, \( N \) is the number of cores and \( P \) is the percentage of parallelizable code.
If we use the same example as before, the scaled speedup calculated by the Gustafson law is:

\[ \text{Speedup} = 4 - 0.25 \times (3) = 3.25 \]

**Conclusion**

In this section, you learned some important issues you have to take into account when you want to parallelize a sequential algorithm.

First of all, not every algorithm can be parallelized. For example, if you have to execute a loop where the result of an iteration depends on the result of the previous iteration, you can't parallelize that loop. Recurrent algorithms are another example of algorithms that can be parallelized for a similar reason.

Another important thing you have to keep in mind is that the sequential version of an algorithm with better performance can be a bad starting point to parallelize it. If you start parallelizing an algorithm and you find yourself in trouble because you don't easily find independent portions of the code, you have to look for other versions of the algorithm and verify that the version can be parallelized in an easier way.

Finally, when you implement a concurrent application (from scratch or based on a sequential algorithm), you must take into account the following points:

- **Efficiency**: The parallel algorithm must end in less time than the sequential algorithm. The first goal of parallelizing an algorithm is that its running time is less than the sequential one, or it can process more data in the same time.

- **Simplicity**: When you implement an algorithm (parallel or not), you must keep it as simple as possible. It will be easier to implement, test, debug, and maintain, and it will have fewer errors.

- **Portability**: Your parallel algorithm should be executed on different platforms with minimal changes. As in this book you will use Java, this point will be very easy. With Java, you can execute your programs in every operating system without any change (if you implement the program as you must).

- **Scalability**: What happens to your algorithm if you increase the number of cores? As mentioned before, you should use all the available cores, so your algorithm should be ready to take advantage of all available resources.
Java concurrency API
The Java programming language has a very rich concurrency API. It contains classes to manage the basic elements of concurrency, such as Thread, Lock, and Semaphore, and classes that implement very high-level synchronization mechanisms, such as the executor framework or the new parallel Stream API.

In this section, we will cover the basic classes that form the concurrency API.

Basic concurrency classes
The basic classes of the Java concurrency API are:

- The Thread class: This class represents all the threads that execute a concurrent Java application
- The Runnable interface: This is another way to create concurrent applications in Java
- The ThreadLocal class: This is a class to store variables locally to a thread
- The ThreadFactory interface: This is the base of the Factory design pattern that you can use to create customized threads

Synchronization mechanisms
The Java concurrency API includes different synchronization mechanisms that allow you to:

- Define a critical section to access a shared resource
- Synchronize different tasks in a common point

The following mechanisms are considered to be the most important synchronization mechanisms:

- The synchronized keyword: The synchronized keyword allows you to define a critical section in a block of code or in an entire method.
- The Lock interface: Lock provides a more flexible synchronization operation than the synchronized keyword. There are different kinds of Locks: ReentrantLock, to implement a Lock that can be associated with a condition; ReentrantReadWriteLock, which separates read and write operations; and StampedLock, a new feature of Java 8 that includes three modes for controlling read/write access.
The Semaphore class: The class that implements the classical semaphore to implement synchronization. Java supports binary and general semaphores.

The CountDownLatch class: A class that allows a task to wait for the finalization of multiple operations.

The CyclicBarrier class: A class that allows the synchronization of multiple threads in a common point.

The Phaser class: A class that allows you to control the execution of tasks divided into phases. None of the tasks advance to the next phase until all of the tasks have finished the current phase.

**Executors**

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:

- The Executor and ExecutorService interface: They include methods common to all executors.
- ThreadPoolExecutor: This is a class that allows you to get an executor with a pool of threads and optionally define a maximum number of parallel tasks
- ScheduledThreadPoolExecutor: This is a special kind of executor to allow you to execute tasks after a delay or periodically
- Executors: This is a class that facilitates the creation of executors
- The Callable interface: This is an alternative to the Runnable interface—a separate task that can return a value
- The Future interface: This is an interface that includes the methods to obtain the value returned by a Callable interface and to control its status
The Fork/Join framework

The Fork/Join framework defines a special kind of executor specialized in the resolution of problems with the divide and conquer technique. It includes a mechanism to optimize the execution of the concurrent tasks that solve these kinds of problems. Fork/Join is specially tailored for fine-grained parallelism as it has a very low overhead in order to place the new tasks into the queue and take queued tasks for execution. The main classes and interfaces involved in this framework are:

- ForkJoinPool: This is a class that implements the executor that is going to run the tasks
- ForkJoinTask: This is a task that can be executed in the ForkJoinPool class
- ForkJoinWorkerThread: This is a thread that is going to execute tasks in the ForkJoinPool class

Parallel streams

Streams and Lambda expressions are maybe the two most important new features of the Java 8 version. Streams have been added as a method in the Collection interface and other data sources and allow processing all elements of a data structure, generating new structures, filtering data and implementing algorithms using the map and reduce technique.

A special kind of stream is a parallel stream which realizes its operations in a parallel way. The most important elements involved in the use of parallel streams are:

- The Stream interface: This is an interface that defines all the operations that you can perform on a stream.
- Optional: This is a container object that may or may not contain a non-null value.
- Collectors: This is a class that implements reduction operations that can be used as part of a stream sequence of operations.
- Lambda expressions: Streams has been thought to work with Lambda expressions. Most stream methods accept a lambda expression as a parameter. This allows you to implement a more compact version of the operations.
Concurrent data structures

Normal data structures of the Java API (ArrayList, Hashtable, and so on) are not ready to work in a concurrent application unless you use an external synchronization mechanism. If you use it, you will be adding a lot of extra computing time to your application. If you don't use it, it's probable that you will have race conditions in your application. If you modify them from several threads and a race condition occurs, you may experience various exceptions thrown (such as, ConcurrentModificationException and ArrayIndexOutOfBoundsException), there may be silent data loss or your program may even stuck in an endless loop.

The Java concurrency API includes a lot of data structures that can be used in concurrent applications without risk. We can classify them in two groups:

- **Blocking data structures**: These include methods that block the calling task when, for example, the data structure is empty and you want to get a value.
- **Non-blocking data structures**: If the operation can be made immediately, it won't block the calling tasks. Otherwise, it returns the null value or throws an exception.

These are some of the data structures:

- ConcurrentLinkedDeque: This is a non-blocking list
- ConcurrentLinkedQueue: This is a non-blocking queue
- LinkedBlockingDeque: This is a blocking list
- LinkedBlockingQueue: This is a blocking queue
- PriorityBlockingQueue: This is a blocking queue that orders its elements based on its priority
- ConcurrentHashMap: This is a non-blocking hash map
- AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference: These are atomic implementations of the basic Java data types

Concurrency design patterns

In software engineering, a design pattern is a solution to a common problem. This solution has been used many times, and it has proved to be an optimal solution to the problem. You can use them to avoid 'reinventing the wheel' every time you have to solve one of these problems. Singleton or Factory are the examples of common design patterns used in almost every application.
Concurrency also has its own design patterns. In this section, we describe some of the most useful concurrency design patterns and their implementation in the Java language.

**Signaling**
This design pattern explains how to implement the situation where a task has to notify an event to another task. The easiest way to implement this pattern is with a semaphore or a mutex, using the `ReentrantLock` or `Semaphore` classes of the Java language or even the `wait()` and `notify()` methods included in the `Object` class.

See the following example:

```java
public void task1() {
    section1();
    commonObject.notify();
}

public void task2() {
    commonObject.wait();
    section2();
}
```

Under these circumstances, the `section2()` method will always be executed after the `section1()` method.

**Rendezvous**
This design pattern is a generalization of the **Signaling** pattern. In this case, the first task waits for an event of the second task and the second task waits for an event of the first task. The solution is similar to that of Signaling, but in this case you must use two objects instead of one.

See the following example:

```java
public void task1() {
    section1_1();
    commonObject1.notify();
    commonObject2.wait();
    section1_2();
}

public void task2() {
    section2_1();
    commonObject2.notify();
    commonObject1.wait();
    section2_2();
}
```
Under these circumstances, section2_2() always will be executed after section1_1() and section1_2() after section2_1(), take into account that, if you put the call to the wait() method before the call to the notify() method, you will have a deadlock.

**Mutex**

A mutex is a mechanism that you can use to implement a critical section ensuring mutual exclusion. That is to say, only one task can execute the portion of code protected by the mutex at one time. In Java, you can implement a critical section using the synchronized keyword (that allows you to protect a portion of code or a full method), the ReentrantLock class, or the Semaphore class.

Look at the following example:

```java
public void task() {
    preCriticalSection();
    lockObject.lock(); // The critical section begins
    criticalSection();
    lockObject.unlock(); // The critical section ends
    postCriticalSection();
}
```

**Multiplex**

The **Multiplex design pattern** is a generalization of the mutex. In this case, a determined number of tasks can execute the critical section at once. It is useful, for example, when you have multiple copies of a resource. The easiest way to implement this design pattern in Java is using the Semaphore class initialized to the number of tasks that can execute the critical section at once.

Look at the following example:

```java
public void task() {
    preCriticalSection();
    semaphoreObject.acquire();
    criticalSection();
    semaphoreObject.release();
    postCriticalSection();
}
```
The First Step – Concurrency Design Principles

Barrier
This design pattern explains how to implement the situation where you need to synchronize some tasks at a common point. None of the tasks can continue with their execution until all the tasks have arrived at the synchronization point. The Java concurrency API provides the CyclicBarrier class, which is an implementation of this design pattern.

Look at the following example:

```java
public void task() {
    preSyncPoint();
    barrierObject.await();
    postSyncPoint();
}
```

Double-checked locking
This design pattern provides a solution to the problem that occurs when you acquire a lock and then check for a condition. If the condition is false, you have had the overhead of acquiring the lock ideally. An example of this situation is the lazy initialization of objects. If you have a class implementing the Singleton design pattern, you may have some code like this:

```java
public class Singleton{
    private static Singleton reference;
    private static final Lock lock=new ReentrantLock();
    public static Singleton getReference() {
        lock.lock();
        try {
            if (reference==null) {
                reference=new Object();
            }
        } finally {
            lock.unlock();
        }
        return reference;
    }
}
```

A possible solution can be to include the lock inside the conditions:

```java
public class Singleton{
    private Object reference;
    private Lock lock=new ReentrantLock();
    public Object getReference() {
        if (reference==null) {
            lock.lock();
            reference=new Object();
            lock.unlock();
        }
        return reference;
    }
}
```
try {
    if (reference == null) {
        reference = new Object();
    }
} finally {
    lock.unlock();
}

return reference;

This solution still has problems. If two tasks check the condition at once, you will create two objects. The best solution to this problem doesn't use any explicit synchronization mechanism:

```java
public class Singleton {

    private static class LazySingleton {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getSingleton() {
        return LazySingleton.INSTANCE;
    }
}
```

**Read-write lock**

When you protect access to a shared variable with a lock, only one task can access that variable, independently of the operation you are going to perform on it. Sometimes, you will have variables that you modify a few times but read many times. In this circumstance, a lock provides poor performance because all the read operations can be made concurrently without any problem. To solve this problem, there exists the read-write lock design pattern. This pattern defines a special kind of lock with two internal locks: one for read operations and the other for write operations. The behavior of this lock is as follows:

- If one task is doing a read operation and another task wants to do another read operation, it can do it
- If one task is doing a read operation and another task wants to do a write operation, it's blocked until all the readers finish
- If one task is doing a write operation and another task wants to do an operation (read or write), it's blocked until the writer finishes
The Java concurrency API includes the class `ReentrantReadWriteLock` that implements this design pattern. If you want to implement this pattern from scratch, you have to be very careful with the priority between read-tasks and write-tasks. If too many read-tasks exist, write-tasks can be waiting too long.

**Thread pool**

This design pattern tries to remove the overhead introduced by creating a thread for the task you want to execute. It's formed by a set of threads and a queue of tasks you want to execute. The set of threads usually has a fixed size. When a thread approaches the execution of a task, it doesn't finish its execution; it looks for another task in the queue. If there is another task, it executes it. If not, the thread waits until a task is inserted in the queue, but it's not destroyed.

The Java concurrency API includes some classes that implement the `ExecutorService` interface, which internally uses a pool of threads.

**Thread local storage**

This design pattern defines how to use global or static variables locally to tasks. When you have a static attribute in a class, all the objects of a class access the same occurrences of the attribute. If you use thread local storage, each thread accesses a different instance of the variable.

The Java concurrency API includes the `ThreadLocal` class to implement this design pattern.

**The Java memory model**

When you execute a concurrent application in a computer with several cores or processors, you can have a problem with memory caches. They are very useful to increment the performance of the application, but they can cause data inconsistency. When a task modifies the value of a variable, it's modified in the cache, but it's not modified in the main memory immediately. If another task reads the value of that variable before it's updated in the main memory, it will read the old value of the variable.

Other problems that may exist with concurrent applications are the optimizations introduced by the compilers and code optimizer. Sometimes, they reorder the instructions to get a better performance. In sequential applications, this doesn't cause any problems, but in concurrent applications it can provoke unexpected results.
To solve problems such as this, programming languages introduced memory models. A memory model describes how individual tasks interact with each other through memory and when changes made by one task will be visible to another. It also defines what optimizations of code are allowed and under what circumstances.

There are different memory models. Some of them are very strict (all of the tasks always have access to the same values) and others are less stringent (only some instructions update the values in the main memory). The memory model must be known by the compiler and optimizer developers, and it's transparent to the rest of the programmers.

Java was the first programming language that defined its memory model. The original memory model defined in the JVM had some issues, and it was redefined in Java 5. That memory model is the same in Java 8. It's defined in JSR 133. Basically, the Java Memory Model defines the following:

- It defines the behavior of the volatile, synchronized, and final keywords.
- It ensures that a properly synchronized concurrent program runs correctly on all architectures.
- It creates a partial ordering of the volatile read, volatile write, lock, and unlock instructions denominated as happens-before. Task synchronization helps us establish the happens-before relations too. If one action happens-before another, then the first is visible to and ordered before the second.
- When a task acquires a monitor, the memory cache is invalidated.
- When a task releases a monitor, the cache data is flushed into the main memory.
- It's transparent for Java programmers.

The main objective of the Java memory model is that the properly written concurrent application will behave correctly on every Java Virtual Machine (JVM) regardless of operating system, CPU architecture, and the number of CPUs and cores.

**Tips and tricks to design concurrent algorithms**

In this section, we have compiled some tips and tricks you have to keep in mind to design good concurrent applications.
The First Step – Concurrency Design Principles

Identify the correct independent tasks
You can only execute concurrent tasks that are independent of each other. If you have two or more tasks with an order dependency between them, maybe you have no interest in trying to execute them concurrently and including a synchronization mechanism to guarantee the execution order. The tasks will execute in a sequential way, and you will have to overcome the synchronization mechanism. A different situation is when you have a task with some prerequisites, but these prerequisites are independent of each other. In this case, you can execute the prerequisites concurrently and then use a synchronization class to control the execution of the task after completion of all the prerequisites.

Another situation where you can't use concurrency is when you have a loop, and all the steps use data generated in the step before or there is some status information that goes from one step to the next step.

Implement concurrency at the highest possible level
Rich threading APIs, as the Java concurrency API, offer you different classes to implement concurrency in your applications. In the case of Java, you can control the creation and synchronization of threads using the Thread or Lock classes, but it also offers you high-level concurrency objects, such as executors or the Fork/Join framework that allows you to execute concurrent tasks. This high-level mechanism offers you the following benefits:

• You don’t have to worry about the creation and management of threads. You only create tasks and send them for execution. The Java concurrency API controls the creation and management of threads for you.
• They are optimized to give better performance than using threads directly. For example, they use a pool of threads to reuse them and avoid thread creation for every task. You can implement these mechanisms from scratch, but it will take you a lot of time, and it will be a complex task.
• They include advanced features that make the API more powerful. For example, with executors in Java, you can execute tasks that return a result in the form of a Future object. Again, you can implement these mechanisms from scratch, but it's not advisable.
• Your application will be migrated easier from one operating system to another, and it will be more scalable.
• Your application might become faster in the future Java versions. Java developers constantly improve the internals, and JVM optimizations will be likely more tailored for JDK APIs.

In summary, for performance and development time reasons, analyze the high-level mechanisms your thread API offers you before implementing your concurrent algorithm.

Take scalability into account

One of the main objectives when you implement a concurrent algorithm is to take advantage of all the resources of your computer, especially the number of processors or cores. But this number may change over time. Hardware is constantly evolving and its cost becomes lower each year.

When you design a concurrent algorithm using data decomposition, don't presuppose the number of cores or processors that your application will execute on. Get the information about the system dynamically (for example, in Java you can get it with the method `Runtime.getRuntime().availableProcessors()`) and make your algorithm use that information to calculate the number of tasks it's going to execute. This process will have an overhead over the execution time of your algorithm, but your algorithm will be more scalable.

If you design a concurrent algorithm using task decomposition, the situation can be more difficult. You depend on the number of independent tasks you have in the algorithm and forcing a bigger number of tasks will increment the overhead introduced by synchronization mechanisms, and the global performance of the application can be even worse. Analyze in detail the algorithm to determine whether you can have a dynamic number of tasks or not.

Use thread-safe APIs

If you need to use a Java library in a concurrent application, read its documentation first to know if it's thread-safe or not. If it's thread-safe, you can use it in your application without any problem. If it's not, you have the following two options:

• If a thread-safe alternative exists, you should use it
• If a thread-safe alternative doesn't exist, you should add the necessary synchronization to avoid all possible problematic situations, especially data race conditions
The First Step – Concurrency Design Principles

For example, if you need a List in a concurrent application, you should not use the ArrayList class if you are going to update it from several threads because it's not thread-safe. In this case, you can use a thread-safe class such as ConcurrentLinkedDeque, CopyOnWriteArrayList, or LinkedBlockingDeque. If the class you want to use is not thread-safe, first you must look for a thread-safe alternative. Probably, it will be more optimized to work with concurrency that any alternative that you can implement.

**Never assume an execution order**

The execution of tasks in a concurrent application when you don't use any synchronization mechanism is nondeterministic. The order in which the tasks are executed and the time each task is in execution before the processor moves on to another task is determined by the scheduler of the operating system. It doesn't care if you observe that the execution order is the same in a number of executions. The next one could be different.

The result of this assumption used to be a data race problem. The final result of your algorithm depends on the execution order of the tasks. Sometimes, the result can be right, but at other times it can be wrong. It can be very difficult to detect the cause of data race conditions, so you must be careful not to forget all the necessary synchronization elements.

**Prefer local thread variables over static and shared when possible**

Thread local variables are a special kind of variable. Every task will have an independent value for this variable, so you don't need any synchronization mechanism to protect access to this variable.

This can sound a little strange. Every object has its own copy of the attributes of the class, so why do we need the thread local variables? Consider this situation. You create a Runnable task, and you want to execute multiple instances of that task. You can create a Runnable object for each thread you want to execute, but another option is to create a Runnable object and use that object to create all the threads. In the last case, all the threads will have access to the same copy of the attributes of the class except if you use the ThreadLocal class. The ThreadLocal class guarantees you that every thread will access its own instance of the variable without the use of a Lock, a semaphore, or a similar class.
Another situation when you can take advantage of Thread local variables is with static attributes. All instances of a class share the static attributes, but you declare them with the `ThreadLocal` class. In this case, every thread will have access to its own copy.

Another option you have is to use something like `ConcurrentHashMap<Thread, MyType>` and use it like `var.get(Thread.currentThread())` or `var.put(Thread.currentThread(), newValue)`. Usually, this approach is significantly slower than `ThreadLocal` because of possible contention (`ThreadLocal` has no contention at all). It has an advantage though: you can clear the map completely and the value will disappear for every thread; thus, sometimes it's useful to use such an approach.

**Find the more easily parallelizable version of the algorithm**

We can define an algorithm as a sequence of steps to solve a problem. There are different ways to solve the same problem. Some are faster, some use fewer resources, and others fit better with special characteristics of the input data. For example, if you want to order a set of numbers, you can use one of the multiple sorting algorithms that have been implemented.

In a previous section of this chapter, we recommended you use a sequential algorithm as the starting point to implement a concurrent algorithm. There are two main advantages to this approach:

- You can easily test the correctness of the results of your parallel algorithm
- You can measure the improvement in performance obtained with the use of concurrency

But not every algorithm can be parallelized, at least not so easily. You might think that the best starting point could be the sequential algorithm with the best performance solving the problem you want to parallelize, but this can be a wrong assumption. You should look for an algorithm than can be easily parallelized. Then, you can compare the concurrent algorithm with the sequential one with the best performance to see which offers the best throughput.

**Using immutable objects when possible**

One of the main problems you can have in a concurrent application is a data race condition. As we explained before, this happens when two or more tasks modify the data stored in a shared variable and access to that variable is not implemented inside a critical section.
The First Step – Concurrency Design Principles

For example, when you work with an object-oriented language such as Java, you implement your application as a collection of objects. Each object has a number of attributes and some methods to read and change the values of the attributes. If some tasks share an object and call to a method to change a value of an attribute of that object and that method is not protected by a synchronization mechanism, you probably will have data inconsistency problems.

There are special kinds of object named immutable objects. Their main characteristic is that you can't modify any attributes after initialization. If you want to modify the value of an attribute, you must create another object. The `String` class in Java is the best example of immutable objects. When you use an operator (for example, `-` or `+-`) that we might think changes the value of a String, you are really creating a new object.

The use of immutable objects in a concurrent application has two very important advantages:

- You don't need any synchronization mechanism to protect the methods of these classes. If two tasks want to modify the same object, they will create new objects, so it will never occur that two tasks modify the same object at a time.
- You won't have any data inconsistency problems, as a conclusion of the first point.

There is a drawback with immutable objects. If you create too many objects, this may affect the throughput and memory use of the application. If you have a simple object without internal data structures, it's usually not a problem to make it immutable. However, making immutable complex objects that incorporate collections of other objects usually leads to serious performance problems.

**Avoiding deadlocks by ordering the locks**

One of the best mechanisms to avoid a deadlock situation in a concurrent application is to force tasks to get shared resources always in the same order. An easy way to do this is to assign a number to every resource. When a task needs more than one resource, it has to request them in order.

For example, if you have two tasks, T1 and T2, and both need two resources, R1 and R2, you can force both to request first the R1 resource and then the R2 resource. You will never have a deadlock.

On the other hand, if T1 first requests R1 and then R2 and T2 first requests R2 and then R1, you can have a deadlock.
For example, a bad use of this tip is as follows. You have two tasks that need to get two Lock objects. They try to get the locks in different order:

```java
public void operation1() {
    lock1.lock();
    lock2.lock();
    ...
}
public void operation2() {
    lock2.lock();
    lock1.lock();
    ....
}
```

It's possible that `operation1()` executes its first sentence and `operation2()` its first sentence too, so they will be waiting for the other Lock and you will have a deadlock.

You can avoid this simply by getting the locks in the same order. If you change `operation2()`, you will never have a deadlock as follows:

```java
public void operation2() {
    lock1.lock();
    lock2.lock();
    ....
}
```

**Using atomic variables instead of synchronization**

When you have to share data between two or more tasks, you have to use a synchronization mechanism to protect the access to that data and avoid any data inconsistency problems.

Under some circumstances, you can use the `volatile` keyword and not use a synchronization mechanism. If only one of the tasks modifies the data and the rest of the tasks read it, you can use the volatile keyword without any synchronization or data inconsistency problem. In other scenarios, you need to use a lock, the synchronized keyword, or any other synchronization method.
In Java 5, the concurrency API included a new kind of variable called atomic variables. These variables are classes that support atomic operations on single variables. They include a method, denominated by `compareAndSet(oldValue, newValue)`, that includes a mechanism to detect if assigning to the new value to the variable is done in one step. If the value of the variable is equal to `oldValue`, it changes it to `newValue` and returns `true`. Otherwise, it returns `false`. There are more methods that work in a similar way, such as `getAndIncrement()` or `getAndDecrement()`. These methods are also atomic.

This solution is lock-free; that is to say, it doesn't use locks or any synchronization mechanism, so its performance is better than any synchronized solution.

The most important atomic variables that you can use in Java are:

- AtomicInteger
- AtomicLong
- AtomicReference
- AtomicBoolean
- LongAdder
- DoubleAdder

**Holding locks for as short a time as possible**

Locks, as with any other synchronization mechanism, allow you to define a critical section that only one task can execute at a time. While a task is executing the critical section, the other tasks that want to execute it are blocked and have to wait for the liberation of the critical section. The application is working in a sequential way.

You have to pay special attention to the instructions you include in your critical sections because you can degrade the performance of your application without realizing it. You must make your critical section as small as possible, and it must include only the instructions that work on shared data with other tasks, so the time that the application is executing in a sequential way will be minimal.
Avoid executing inside the critical section the code you don't control. For example, you are writing a library that accepts a user-defined Callable, which you need to launch sometimes. You don't know what exactly will be in that Callable. Maybe it blocks input/output, acquires some locks, calls other methods of your library, or just works for a very long time. Thus, whenever possible, try to execute it when your library does not hold any locks. If it's impossible for your algorithm, specify this behavior in your library documentation and possibly specify the limitations to the user-supplied code (for example, it should not take any locks). A good example of such documentation can be found in the `compute()` method of the `ConcurrentHashMap` class.

**Taking precautions using lazy initialization**

Lazy initialization is a mechanism that delays object creation until the object is used in the application for the first time. Its main advantage is it minimizes the use of memory because you only create the objects that are really needed, but it can be a problem in concurrent applications.

If you have a method that initializes an object and this method is called by two different tasks at once, you can initialize two different objects. This, for example, can be a problem with singleton classes because you only want to create one object of these classes.

A elegant solution to this problem has been implemented, as the Initialization-on-demand holder idiom ([https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom](https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom)).

**Avoiding the use of blocking operations inside a critical section**

Blocking operations are those operations that block the task that calls them until an event occurs. For example, when you read data from a file or write data to the console, the task that calls these operations must wait until they finish.

If you include one of these operations into a critical section, you are degrading the performance of your application because none of the tasks that want to execute that critical section can execute it. The one that is inside the critical section is waiting for the finalization of an I/O operation, and the others are waiting for the critical section.

Unless it is imperative, don't include blocking operations inside a critical section.
Summary

Concurrent programming includes all the necessary tools and techniques to have multiple tasks or process running at the same time in a computer, communicating and synchronizing between them without data loss or inconsistency.

We started this chapter by introducing the basic concepts of concurrency. You must know and understand terms such as concurrency, parallelism, and synchronization to fully understand the examples of this book. However, concurrency can generate some problems, such as data race conditions, deadlocks, livelocks, and others. You must also know the potential problems of a concurrent application. It will help you identify and solve these problems.

We also explained a simple methodology of five steps introduced by Intel to convert a sequential algorithm into a concurrent one and showed you some concurrency design patterns implemented in the Java language and some tips to take into account when you implement a concurrent application.

Finally, we explained briefly the components of the Java concurrency API. It's a very rich API with low- and very high-level mechanisms that allow you to implement powerful concurrency applications easily. We also described the Java memory model, which determines how concurrent applications manage the memory and the execution order of the instructions internally.

In the next chapter, you will learn how to implement applications that use a lot of threads using the executor framework. This allows you to execute a big number of threads by controlling the resources you use and reducing the overhead introduced by thread creation (it reuses Thread objects to execute different tasks).
Where to buy this book

You can buy Mastering Concurrency Programming with Java 8 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.