Quick answers to common problems

# Java 7 Concurrency Cookbook

Over 60 simple but incredibly effective recipes for mastering multithreaded application development with Java 7

Javier Fernández González

[PACKT] PUBLISHING enterprise professional expertise distilled

# Java 7 Concurrency Cookbook

Over 60 simple but incredibly effective recipes for mastering multithreaded application development with Java 7

**Javier Fernández González**

[PACKT] PUBLISHING enterprise
professional expertise distilled

BIRMINGHAM - MUMBAI

# Java 7 Concurrency Cookbook



First published: October 2012

Production Reference: 1181012

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-788-1

`www.packtpub.com`

Cover Image by Artie Ng (`artherng@yahoo.com.au`)

# Credits

**Author**
Javier Fernández González

**Reviewers**
Edward E. Griebel Jr
Jacek Laskowski
Abraham Tehrani

**Acquisition Editor**
Stephanie Moss

**Lead Technical Editor**
Unnati Shah

**Technical Editors**
Kaustubh S. Mayekar
Ankita Meshram

**Copy Editors**
Alfida Paiva
Laxmi Subramanian

**Project Coordinator**
Leena Purkait

**Proofreaders**
Linda Morris
Lauren Tobon

**Indexer**
Hemangini Bari

**Production Coordinator**
Melwyn D'Sa

**Cover Work**
Melwyn D'Sa

# About the Author

**Javier Fernández González** is a software architect with over 10 years experience with Java technologies. He has worked as a teacher, researcher, programmer, analyst, 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 for processing large amount of data in Java and has participated as a co-author on several journal articles and conference presentations. 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). He currently works as a software architect at Capgemini developing and maintaining applications for an insurance company.

# About the Reviewers

**Edward E. Griebel Jr**'s first introduction to computers was in elementary school through LOGO on an Apple and The Oregon Trail on a VAX. Pursuing his interest in computers, he graduated from Bucknell University with a degree in Computer Engineering. At his first job, he quickly realized he didn't know everything that there was to know about computer programming. He has spent the past 20 years honing his skills in the securities trading, telecommunications, payroll processing, and machine-to-machine communications industries as a developer, team leader, consultant, and mentor. Currently working on enterprise development in Java EE, he feels that any day spent writing a code is a good day.

I would like to thank my wife and three children who are used to letting me sleep late after long nights at the computer.

**Jacek Laskowski** is a professional software specialist using a variety of commercial and open source solutions to meet customer's demands. He develops applications, writes articles, guides less-experienced engineers, records screen casts, delivers courses, and has been a technical reviewer for many IT books.

He focuses on Java EE, Service-Oriented Architecture (SOA), Business Process Management (BPM) solutions, OSGi, and functional languages (Clojure and F#). He's into Scala, Dart, native Android development in Java and HTML 5.

He is the founder and leader of the Warszawa Java User Group (Warszawa JUG). He is also a member of the Apache Software Foundation, and a PMC and committer of Apache OpenEJB and Apache Geronimo projects.

He regularly speaks at developer conferences. He blogs at `http://blog.japila.pl` and `http://blog.jaceklaskowski.pl`. Follow him on twitter `@jaceklaskowski`.

He has been working for IBM for over 6 years now and is currently a Certified IT Specialist (Level 2) in the World-wide Web Sphere Competitive Migration Team. He assists customers in their migrations from competitive offerings, mostly Oracle WebLogic Server, to the IBM WebSphere Application Server.

He's recently been appointed to the IBM Academy of Technology.

I'd like to thank my family – my wife Agata, and 3 kids Iweta, Patryk, and Maksym, for their constant support, encouragement, and patience. Without you, I wouldn't have achieved so much! Love you all immensely.

**Abraham Tehrani**, over a decade, has software development experience as a developer and QA engineer. Also, he is passionate about quality and technology.

I would like to thank my fiancé for her support and love and my friends and family for supporting me in all of my endeavors.

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLIB

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

## Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

# Table of Contents

# Preface

When you work with a computer, you can do several things at once. You can hear music while you edit a document in a word processor and read your e-mail. This can be done because your operating system allows the concurrency of tasks. Concurrent programming is about the elements and mechanisms a platform offers to have multiple tasks or programs running at once and communicate with each other to exchange data or to synchronize with each other. Java is a concurrent platform and offers a lot of classes to execute concurrent tasks inside a Java program. With each version, Java increases the functionalities offered to programmers to facilitate the development of concurrent programs. This book covers the most important and useful mechanisms included in Version 7 of the Java concurrency API, so you will be able to use them directly in your applications, which are as follows:

- Basic thread management
- Thread synchronization mechanisms
- Thread creation and management delegation with executors
- Fork/Join framework to enhance the performance of your application
- Data structures for concurrent programs
- Adapting the default behavior of some concurrency classes to your needs
- Testing Java concurrency applications

## What this book covers

*Chapter 1*, *Thread Management* will teach the readers how to make basic operations with threads. Creation, execution, and status management of the threads are explained through basic examples.

*Chapter 2*, *Basic Thread Synchronization* will teach the readers to use the low-level Java mechanisms to synchronize a code. Locks and the `synchronized` keyword are explained in detail.

*Chapter 3*, *Thread Synchronization Utilities* will teach the readers to use the high-level utilities of Java to manage the synchronization between the threads in Java. It includes an explanation of how to use the new Java 7 `Phaser` class to synchronize tasks divided into phases.

*Chapter 4*, *Thread Executors* will teach the readers to delegate the thread management to executors. They allow running, managing, and getting the results of concurrent tasks.

*Chapter 5*, *Fork/Join Framework* will teach the readers to use the new Java 7 Fork/Join framework. It's a special kind of executor oriented to execute tasks that will be divided into smaller ones using the divide and conquer technique.

*Chapter 6*, *Concurrent Collections* will teach the readers to how to use some concurrent data structures provided by the Java language. These data structures must be used in concurrent programs to avoid the use of synchronized blocks of code in their implementation.

*Chapter 7*, *Customizing Concurrency Classes* will teach the readers how to adapt some of the most useful classes of the Java concurrency API to their needs.

*Chapter 8*, *Testing Concurrent Applications* will teach the readers how to obtain information about the status of some of the most useful structures of the Java 7 concurrency API. The readers will also learn how to use some free tools to debug concurrent applications, such as the Eclipse, NetBeans IDE, or FindBugs applications to detect possible bugs on their applications.

*Chapter 9*, *Additional Information* is not present in the book but is available as a free download from the following link: `http://www.packtpub.com/sites/default/files/downloads/Additional`

This chapter will teach the readers the notions of synchronization, the Executor, and Fork/Join frameworks, concurrent data structures, and monitoring of concurrent objects that was not included in the respective chapters.

*Appendix*, *Concurrent Programming Design* is not present in the book but is available as a free download from the following link: `http://www.packtpub.com/sites/default/files/downloads/Concurrent`

This appendix will teach the readers some tips that every programmer should consider when he or she is going to develop a concurrent application.

# What you need for this book

To follow this book, you need a basic knowledge of the Java programming language. You should know how to use an IDE, such as Eclipse or NetBeans, but this is not a necessary prerequisite.

## Who this book is for

If you are a Java developer, who wants to take his knowledge of concurrent programming and multithreading further, as well as discover the new concurrency features of Java 7, then *Java 7 Concurrency Cookbook* is for you. You should already be comfortable with general Java development practices and a basic grasp of threads would be an advantage.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Extending the `Thread` class and overriding the `run()` method".

A block of code is set as follows:

```
public Calculator(int number) {
  this.number=number;
}
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Create a new project with the **New Project** option of the **File** menu in the menu bar".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1
# Thread Management

In this chapter, we will cover:

- Creating and running a thread
- Getting and setting thread information
- Interrupting a thread
- Controlling the interruption of a thread
- Sleeping and resuming a thread
- Waiting for the finalization of a thread
- Creating and running a daemon thread
- Processing uncontrolled exceptions in a thread
- Using local thread variables
- Grouping threads into a group
- Processing uncontrolled exceptions in a group of threads
- Creating threads through a factory

## Introduction

In the computer world, when we talk about **concurrency**, we talk about a series of tasks that run simultaneously in a computer. This simultaneity can be real if the computer has more than one processor or a multi-core processor, or apparent if the computer has only one core processor.

All modern operating systems allow the execution of concurrent tasks. You can read your e-mails while you listen to music and read the news in a web page. We can say that this kind of concurrency is a **process-level** concurrency. But inside a process, we can also have various simultaneous tasks. The concurrent tasks that run inside a process are called **threads**.

Another concept related to concurrency is **parallelism**. There are different definitions and relations with the concurrency concept. Some authors talk about concurrency when you execute your application with multiple threads in a single-core processor, so simultaneously you can see when your program execution is apparent. Also, you can talk about parallelism when you execute your application with multiple threads in a multi-core processor or in a computer with more than one processor. Other authors talk about concurrency when the threads of the application are executed without a predefined order, and talk about parallelism when you use various threads to simplify the solution of a problem, where all these threads are executed in an ordered way.

This chapter presents a number of recipes that show how to perform basic operations with threads using the Java 7 API. You will see how to create and run threads in a Java program, how to control their execution, and how to group some threads to manipulate them as a unit.

# Creating and running a thread

In this recipe, we will learn how to create and run a thread in a Java application. As with every element in the Java language, threads are **objects**. We have two ways of creating a thread in Java:

- Extending the `Thread` class and overriding the `run()` method
- Building a class that implements the `Runnable` interface and then creating an object of the `Thread` class passing the `Runnable` object as a parameter

In this recipe, we will use the second approach to create a simple program that creates and runs 10 threads. Each thread calculates and prints the multiplication table of a number between one and 10.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or another IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Calculator` that implements the `Runnable` interface.

   ```
   public class Calculator implements Runnable {
   ```

2. Declare a `private int` attribute named `number` and implement the constructor of the class that initializes its value.

```
  private int number;
  public Calculator(int number) {
    this.number=number;
  }
```

3. Implement the `run()` method. This method will execute the instructions of the thread that we are creating, so this method will calculate the multiplication table of the number.

```
  @Override
  public void run() {
    for (int i=1; i<=10; i++){
      System.out.printf("%s: %d * %d = %d\n",Thread.
currentThread().getName(),number,i,i*number);
    }
  }
```

4. Now, implement the main class of the application. Create a class named `Main` that contains the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

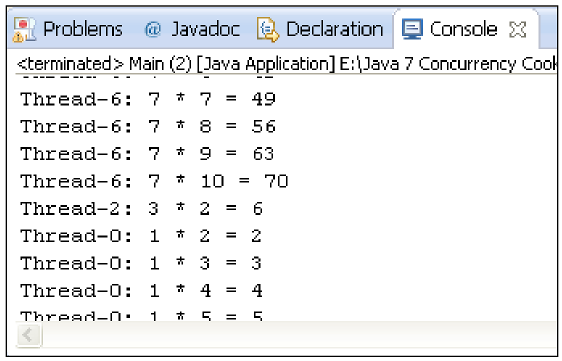
5. Inside the `main()` method, create a `for` loop with 10 iterations. Inside the loop, create an object of the `Calculator` class, an object of the `Thread` class, pass the `Calculator` object as a parameter, and call the `start()` method of the thread object.

```
    for (int i=1; i<=10; i++){
      Calculator calculator=new Calculator(i);
      Thread thread=new Thread(calculator);
      thread.start();
    }
```

6. Run the program and see how the different threads work in parallel.

## How it works...

The following screenshot shows part of the output of the program. We can see that all the threads we have created, run in parallel to do their job, as shown in the following screenshot:




Every Java program has at least one execution thread. When you run the program, the JVM runs this execution thread that calls the `main()` method of the program.

When we call the `start()` method of a `Thread` object, we are creating another execution thread. Our program will have as many execution threads as calls to the `start()` method are made.

A Java program ends when all its threads finish (more specifically, when all its non-daemon threads finish). If the initial thread (the one that executes the `main()` method) ends, the rest of the threads will continue with their execution until they finish. If one of the threads use the `System.exit()` instruction to end the execution of the program, all the threads end their execution.

Creating an object of the `Thread` class doesn't create a new execution thread. Also, calling the `run()` method of a class that implements the `Runnable` interface doesn't create a new execution thread. Only calling the `start()` method creates a new execution thread.

## There's more...

As we mentioned in the introduction of this recipe, there is another way of creating a new execution thread. You can implement a class that extends the `Thread` class and overrides the `run()` method of this class. Then, you can create an object of this class and call the `start()` method to have a new execution thread.

## See also

- The *Creating threads through a factory* recipe in *Chapter 1*, *Thread Management*

# Getting and setting thread information

The `Thread` class saves some information attributes that can help us to identify a thread, know its status, or control its priority. These attributes are:

- **ID**: This attribute stores a unique identifier for each `Thread`.
- **Name**: This attribute store the name of `Thread`.
- **Priority**: This attribute stores the priority of the `Thread` objects. Threads can have a priority between one and 10, where one is the lowest priority and 10 is the highest one. It's not recommended to change the priority of the threads, but it's a possibility that you can use if you want.
- **Status**: This attribute stores the status of `Thread`. In Java, `Thread` can be in one of these six states: `new`, `runnable`, `blocked`, `waiting`, `time waiting`, or `terminated`.

In this recipe, we will develop a program that establishes the name and priority for 10 threads and then shows information about their status until they finish. The threads will calculate the multiplication table of a number.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Calculator` and specify that it implements the `Runnable` interface.

   ```
   public class Calculator implements Runnable {
   ```

2. Declare an `int private` attribute named `number` and implement the constructor of the class that initializes this attribute.

```
  private int number;
  public Calculator(int number) {
    this.number=number;
  }
```

3. Implement the `run()` method. This method will execute the instructions of the thread that we are creating, so this method will calculate and print the multiplication table of a number.

```
  @Override
  public void run() {
    for (int i=1; i<=10; i++){
      System.out.printf("%s: %d * %d = %d\n",Thread.
currentThread().getName(),number,i,i*number);
    }
  }
```

4. Now, we implement the main class of this example. Create a class named `Main` and implement the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

5. Create an array of 10 `threads` and an array of 10 `Thread.State` to store the threads we are going to execute and their status.

```
    Thread threads[]=new Thread[10];
    Thread.State status[]=new Thread.State[10];
```

6. Create 10 objects of the `Calculator` class, each initialized with a different number, and 10 `threads` to run them. Set the priority of five of them to the maximum value and set the priority of the rest to the minimum value.

```
    for (int i=0; i<10; i++){
      threads[i]=new Thread(new Calculator(i));
      if ((i%2)==0){
        threads[i].setPriority(Thread.MAX_PRIORITY);
      } else {
        threads[i].setPriority(Thread.MIN_PRIORITY);
      }
      threads[i].setName("Thread "+i);
    }
```

7. Create a `PrintWriter` object to write to a file on the evolution of the status of the threads.

```
    try (FileWriter file = new FileWriter(".\\data\\log.txt");
PrintWriter pw = new PrintWriter(file);){
```

8. Write on this file the status of the 10 `threads`. Now, it becomes `NEW`.

```
      for (int i=0; i<10; i++){
pw.println("Main : Status of Thread "+i+" : "  +
threads[i].getState());
        status[i]=threads[i].getState();
      }
```

9. Start the execution of the 10 `threads`.

```
      for (int i=0; i<10; i++){
        threads[i].start();
      }
```

10. Until the 10 `threads` end, we are going to check their status. If we detect a change in the status of a thread, we write them on the file.

```
      boolean finish=false;
      while (!finish) {
        for (int i=0; i<10; i++){
          if (threads[i].getState()!=status[i]) {
            writeThreadInfo(pw, threads[i],status[i]);
            status[i]=threads[i].getState();
          }
        }
        finish=true;
        for (int i=0; i<10; i++){
finish=finish &&(threads[i].getState()==State.TERMINATED);
        }
      }
```

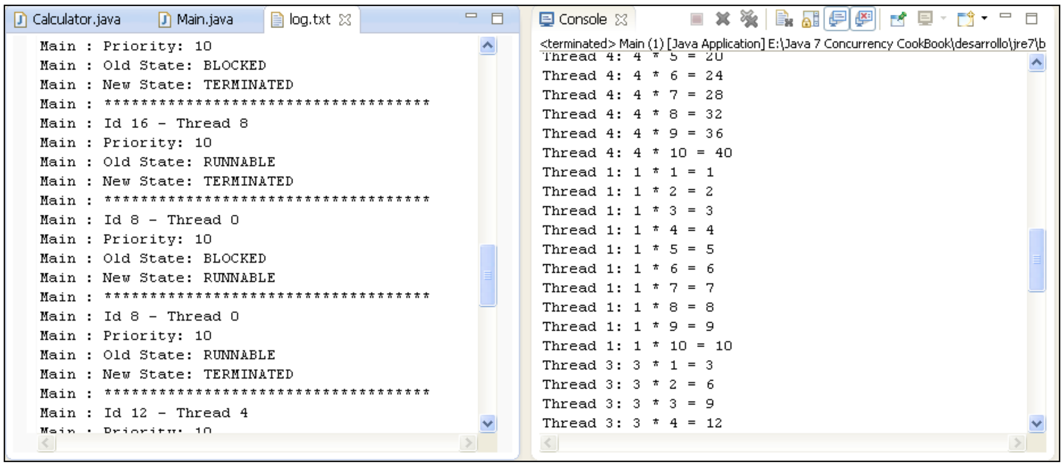
11. Implement the method `writeThreadInfo()` which writes the ID, name, priority, old status, and new status of `Thread`.

```
  private static void writeThreadInfo(PrintWriter pw, Thread
thread, State state) {
pw.printf("Main : Id %d - %s\n",thread.getId(),thread.getName());
pw.printf("Main : Priority: %d\n",thread.getPriority());
pw.printf("Main : Old State: %s\n",state);
pw.printf("Main : New State: %s\n",thread.getState());
pw.printf("Main : ************************************\n");
  }
```

12. Run the example and open the `log.txt` file to see the evolution of the 10 `threads`.

## How it works...

The following screenshot shows some lines of the `log.txt` file in an execution of this program. In this file, we can see that the threads with the highest priority end before the ones with the lowest priority. We also can see the evolution of the status of every thread.




The program shown in the console is the multiplication tables calculated by the threads and the evolution of the status of the different threads in the file `log.txt`. By this way, you can better see the evolution of the threads.

The class `Thread` has attributes to store all the information of a thread. The JVM uses the priority of the threads to select the one that uses the CPU at each moment and actualizes the status of every thread according to its situation.

If you don't specify a name for a thread, the JVM automatically assigns it one with the format, Thread-XX where XX is a number. You can't modify the ID or status of a thread. The `Thread` class doesn't implement the `setId()` and `setStatus()` methods to allow their modification.

## There's more...

In this recipe, you learned how to access the information attributes using a `Thread` object. But you can also access these attributes from an implementation of the `Runnable` interface. You can use the static method `currentThread()` of the `Thread` class to access the `Thread` object that is running the `Runnable` object.

You have to take into account that the `setPriority()` method can throw an `IllegalArgumentException` exception if you try to establish a priority that isn't between one and 10.

## See Also

- The *Interrupting a Thread* recipe in *Chapter 1*, *Thread Management*

# Interrupting a thread

A Java program with more than one execution thread only finishes when the execution of all of its threads end (more specifically, when all its non-daemon threads end its execution or when one of the threads use the `System.exit()` method). Sometimes, you will need to finish a thread, because you want to terminate a program, or when a user of the program wants to cancel the tasks that a `Thread` object is doing.

Java provides the interruption mechanism to indicate to a thread that we want to finish it. One peculiarity of this mechanism is that `Thread` has to check if it has been interrupted or not, and it can decide if it responds to the finalization request or not. `Thread` can ignore it and continue with its execution.

In this recipe, we will develop a program that creates `Thread` and, after 5 seconds, will force its finalization using the interruption mechanism.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `PrimeGenerator` that extends the `Thread` class.

```
public class PrimeGenerator extends Thread{
```

2. Override the `run()` method including a loop that will run indefinitely. In this loop, we are going to process consecutive numbers beginning at one. For each number, we will calculate if it's a prime number and, in that case, we are going to write it to the console.

```
  @Override
  public void run() {
    long number=1L;
    while (true) {
      if (isPrime(number)) {
        System.out.printf("Number %d is Prime",number);
      }
```

3. After processing a number, check if the thread has been interrupted by calling the `isInterrupted()` method. If this method returns `true`, we write a message and end the execution of the thread.

```
      if (isInterrupted()) {
        System.out.printf("The Prime Generator has been 
Interrupted");
        return;
      }
      number++;
    }
  }
```

4. Implement the `isPrime()` method. It returns a `boolean` value indicating if the number that is received as a parameter is a prime number (`true`) or not (`false`).

```
  private boolean isPrime(long number) {
    if (number <=2) {
      return true;
    }
    for (long i=2; i<number; i++){
      if ((number % i)==0) {
        return false;
      }
    }
    return true;
  }
```

5. Now, implement the main class of the example by implementing a class called `Main` and implementing the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

6. Create and start an object of the `PrimeGenerator` class.

```
    Thread task=new PrimeGenerator();
    task.start();
```

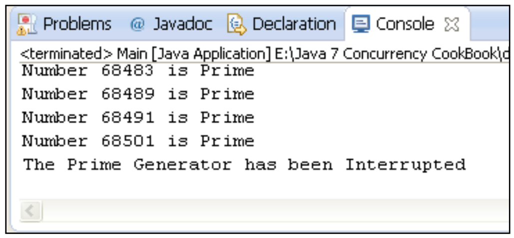
7. Wait for 5 seconds and interrupt the `PrimeGenerator` thread.

```
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
task.interrupt();
```

8. Run the example and see the results.

## How it works...

The following screenshot shows the result of execution of the previous example. We can see how the `PrimeGenerator` thread writes the message and ends its execution when it detects that it has been interrupted. Refer to the following screenshot:




The `Thread` class has an attribute that stores a `boolean` value indicating whether the thread has been interrupted or not. When you call the `interrupt()` method of a thread, you set that attribute to `true`. The `isInterrupted()` method only returns the value of that attribute.

## There's more...

The `Thread` class has another method to check whether `Thread` has been interrupted or not. It's the static method, `interrupted()`, that checks whether the current executing thread has been interrupted or not.

> There is an important difference between the `isInterrupted()` and the `interrupted()` methods. The first one doesn't change the value of the `interrupted` attribute, but the second one sets it to `false`. As the `interrupted()` method is a static method, the utilization of the `isInterrupted()` method is recommended.

As I mentioned earlier, `Thread` can ignore its interruption, but this is not the expected behaviour.

# Controlling the interruption of a thread

In the previous recipe, you learned how you can interrupt the execution of a thread and what you have to do to control this interruption in the `Thread` object. The mechanism shown in the previous example can be used if the thread that can be interrupted is simple. But if the thread implements a complex algorithm divided into some methods, or it has methods with recursive calls, we can use a better mechanism to control the interruption of the thread. Java provides the `InterruptedException` exception for this purpose. You can throw this exception when you detect the interruption of the thread and catch it in the `run()` method.

In this recipe, we will implement `Thread` that looks for files with a determined name in a folder and in all its subfolders to show how to use the `InterruptedException` exception to control the interruption of a thread.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `FileSearch` and specify that it implements the `Runnable` interface.

```
public class FileSearch implements Runnable {
```

2. Declare two `private` attributes, one for the name of the file we are going to search for and one for the initial folder. Implement the constructor of the class, which initializes these attributes.

```
  private String initPath;
  private String fileName;
  public FileSearch(String initPath, String fileName) {
    this.initPath = initPath;
    this.fileName = fileName;
  }
```

3. Implement the `run()` method of the `FileSearch` class. It checks if the attribute `fileName` is a directory and, if it is, calls the method `processDirectory()`. This method can throw an `InterruptedException` exception, so we have to catch them.

```
  @Override
  public void run() {
    File file = new File(initPath);
    if (file.isDirectory()) {
      try {
        directoryProcess(file);
      } catch (InterruptedException e) {
        System.out.printf("%s: The search has been 
interrupted",Thread.currentThread().getName());
      }
    }
  }
```

4. Implement the `directoryProcess()` method. This method will obtain the files and subfolders in a folder and process them. For each directory, the method will make a recursive call passing the directory as a parameter. For each file, the method will call the `fileProcess()` method. After processing all files and folders, the method checks if `Thread` has been interrupted and, in this case, throws an `InterruptedException` exception.

```
  private void directoryProcess(File file) throws
InterruptedException {
    File list[] = file.listFiles();
    if (list != null) {
      for (int i = 0; i < list.length; i++) {
        if (list[i].isDirectory()) {
          directoryProcess(list[i]);
        } else {
          fileProcess(list[i]);
        }
      }
    }
    if (Thread.interrupted()) {
      throw new InterruptedException();
    }
  }
```

5. Implement the `processFile()` method. This method will compare the name of the file it's processing with the name we are searching for. If the names are equal, we will write a message in the console. After this comparison, `Thread` will check if it has been interrupted and, in this case, it throws an `InterruptedException` exception.

```
  private void fileProcess(File file) throws InterruptedException
{
    if (file.getName().equals(fileName)) {
      System.out.printf("%s : %s\n",Thread.currentThread().
getName() ,file.getAbsolutePath());
    }
    if (Thread.interrupted()) {
      throw new InterruptedException();
    }
  }
```

6. Now, let's implement the main class of the example. Implement a class called `Main` that contains the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

7. Create and initialize an object of the `FileSearch` class and `Thread` to execute its task. Then, start executing `Thread`.

```
FileSearch searcher=new FileSearch("C:\\","autoexec.bat");
Thread thread=new Thread(searcher);
thread.start();
```

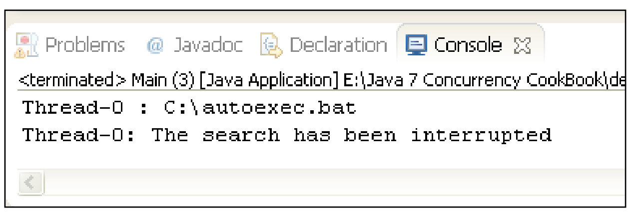
8. Wait for 10 seconds and interrupt `Thread`.

```
  try {
    TimeUnit.SECONDS.sleep(10);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  thread.interrupt();
}
```

9. Run the example and see the results.

## How it works...

The following screenshot shows the result of an execution of this example. You can see how the `FileSearch` object ends its execution when it detects that it has been interrupted. Refer to the following screenshot:




In this example, we use Java exceptions to control the interruption of `Thread`. When you run the example, the program starts going through folders by checking if they have the file or not. For example, if you enter in the folder `\b\c\d`, the program will have three recursive calls to the `processDirectory()` method. When it detects that it has been interrupted, it throws an `InterruptedException` exception and continues the execution in the `run()` method, no matter how many recursive calls have been made.

## There's more...

The `InterruptedException` exception is thrown by some Java methods related with the concurrency API such as `sleep()`.

## See also

- The *Interrupting a thread recipe* in *Chapter 1*, *Thread Management*

# Sleeping and resuming a thread

Sometimes, you'll be interested in interrupting the execution of `Thread` during a determined period of time. For example, a thread in a program checks a sensor state once per minute. The rest of the time, the thread does nothing. During this time, the thread doesn't use any resources of the computer. After this time, the thread will be ready to continue with its execution when the JVM chooses it to be executed. You can use the `sleep()` method of the `Thread` class for this purpose. This method receives an integer as the parameter indicates the number of milliseconds that the thread suspends its execution. When the sleeping time ends, the thread continues with its execution in the instruction, after the `sleep()` method calls, when the JVM assigns them CPU time.

Another possibility is to use the `sleep()` method of an element of the `TimeUnit` enumeration. This method uses the `sleep()` method of the `Thread` class to put the current thread to sleep, but it receives the parameter in the unit that it represents and converts it to milliseconds.

In this recipe, we will develop a program that uses the `sleep()` method to write the actual date every second.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `FileClock` and specify that it implements the `Runnable` interface.

   ```
   public class FileClock implements Runnable {
   ```

2. Implement the `run()` method.

   ```
     @Override
     public void run() {
   ```

3. Write a loop with 10 iterations. In each iteration, create a `Date` object, write it to the file, and call the `sleep()` method of the `SECONDS` attribute of the `TimeUnit` class to suspend the execution of the thread for one second. With this value, the thread will be sleeping for approximately one second. As the `sleep()` method can throw an `InterruptedException` exception, we have to include the code to catch it. It's a good practice to include code that frees or closes the resources the thread is using when it's interrupted.

```
    for (int i = 0; i < 10; i++) {
      System.out.printf("%s\n", new Date());
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        System.out.printf("The FileClock has been interrupted");
      }
    }
  }
```

4. We have implemented the thread. Now, let's implement the main class of the example. Create a class called `FileMain` that contains the `main()` method.

```
public class FileMain {
  public static void main(String[] args) {
```

5. Create an object of the `FileClock` class and a thread to execute it. Then, start executing `Thread`.

```
    FileClock clock=new FileClock();
    Thread thread=new Thread(clock);
    thread.start();
```

6. Call the `sleep()` method of the SECONDS attribute of the `TimeUnit` class in the main `Thread` to wait for 5 seconds.

```
    try {
      TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e) {
      e.printStackTrace();
    };
```

7. Interrupt the `FileClock` thread.

```
    thread.interrupt();
```

8. Run the example and see the results.

## How it works...

When you run the example, you can see how the program writes a `Date` object per second and then, the message indicating that the `FileClock` thread has been interrupted.

When you call the `sleep()` method, `Thread` leaves the CPU and stops its execution for a period of time. During this time, it's not consuming CPU time, so the CPU can be executing other tasks.

When `Thread` is sleeping and is interrupted, the method throws an `InterruptedException` exception immediately and doesn't wait until the sleeping time finishes.

## There's more...

The Java concurrency API has another method that makes a `Thread` object leave the CPU. It's the `yield()` method, which indicates to the JVM that the `Thread` object can leave the CPU for other tasks. The JVM does not guarantee that it will comply with this request. Normally, it's only used for debug purposes.

# Waiting for the finalization of a thread

In some situations, we will have to wait for the finalization of a thread. For example, we may have a program that will begin initializing the resources it needs before proceeding with the rest of the execution. We can run the initialization tasks as threads and wait for its finalization before continuing with the rest of the program.

For this purpose, we can use the `join()` method of the `Thread` class. When we call this method using a thread object, it suspends the execution of the calling thread until the object called finishes its execution.

In this recipe, we will learn the use of this method with the initialization example.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `DataSourcesLoader` and specify that it implements the `Runnable` interface.

```
public class DataSourcesLoader implements Runnable {
```

2. Implement the `run()` method. It writes a message to indicate that it starts its execution, sleeps for 4 seconds, and writes another message to indicate that it ends its execution.

```
  @Override
  public void run() {
    System.out.printf("Beginning data sources loading: %s\n",new 
Date());
    try {
      TimeUnit.SECONDS.sleep(4);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("Data sources loading has finished: 
%s\n",new Date());
  }
```

3. Create a class called `NetworkConnectionsLoader` and specify that it implements the `Runnable` interface. Implement the `run()` method. It will be equal to the `run()` method of the `DataSourcesLoader` class, but this will sleep for 6 seconds.
4. Now, create a class called `Main` that contains the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

5. Create an object of the `DataSourcesLoader` class and `Thread` to run it.

```
    DataSourcesLoader dsLoader = new DataSourcesLoader();
    Thread thread1 = new Thread(dsLoader,"DataSourceThread");
```

6. Create an object of the `NetworkConnectionsLoader` class and `Thread` to run it.

```
    NetworkConnectionsLoader ncLoader = new 
NetworkConnectionsLoader();
    Thread thread2 = new Thread(ncLoader,"NetworkConnectionLoad
er");
```

7. Call the `start()` method of both the `Thread` objects.

```
    thread1.start();
    thread2.start();
```

8. Wait for the finalization of both threads using the `join()` method. This method can throw an `InterruptedException` exception, so we have to include the code to catch it.

```
try {
  thread1.join();
  thread2.join();
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

9. Write a message to indicate the end of the program.

```
    System.out.printf("Main: Configuration has been loaded: 
%s\n",new Date());
```

10. Run the program and see the results.

## How it works...

When you run this program, you can see how both `Thread` objects start their execution. First, the `DataSourcesLoader` thread finishes its execution. Then, the `NetworkConnectionsLoader` class finishes its execution and, at that moment, the main `Thread` object continues its execution and writes the final message.

## There's more...

Java provides two additional forms of the `join()` method:

- join (long milliseconds)
- join (long milliseconds, long nanos)

In the first version of the `join()` method, instead of waiting indefinitely for the finalization of the thread called, the calling thread waits for the milliseconds specified as a parameter of the method. For example, if the object `thread1` has the code, `thread2.join(1000)`, the thread `thread1` suspends its execution until one of these two conditions is true:

- `thread2` finishes its execution
- 1000 milliseconds have been passed

When one of these two conditions is true, the `join()` method returns.

The second version of the `join()` method is similar to the first one, but receives the number of milliseconds and the number of nanoseconds as parameters.

# Creating and running a daemon thread

Java has a special kind of thread called **daemon** thread. These kind of threads have very low priority and normally only executes when no other thread of the same program is running. When daemon threads are the only threads running in a program, the JVM ends the program finishing these threads.

With these characteristics, the daemon threads are normally used as service providers for normal (also called user) threads running in the same program. They usually have an infinite loop that waits for the service request or performs the tasks of the thread. They can't do important jobs because we don't know when they are going to have CPU time and they can finish any time if there aren't any other threads running. A typical example of these kind of threads is the Java garbage collector.

In this recipe, we will learn how to create a daemon thread developing an example with two threads; one user thread that writes events on a queue and a daemon one that cleans that queue, removing the events which were generated more than 10 seconds ago.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create the `Event` class. This class only stores information about the events our program will work with. Declare two private attributes, one called `date` of `java.util.Date` type and the other called `event` of `String` type. Generate the methods to write and read their values.
2. Create the `WriterTask` class and specify that it implements the `Runnable` interface.

   ```
   public class WriterTask implements Runnable {
   ```

3. Declare the queue that stores the events and implement the constructor of the class, which initializes this queue.

   ```
   private Deque<Event> deque;
     public WriterTask (Deque<Event> deque){
       this.deque=deque;
     }
   ```

4. Implement the `run()` method of this task. This method will have a loop with 100 iterations. In each iteration, we create a new `Event`, save it in the queue, and sleep for one second.

```
  @Override
  public void run() {
    for (int i=1; i<100; i++) {
      Event event=new Event();
      event.setDate(new Date());
      event.setEvent(String.format("The thread %s has generated an
event",Thread.currentThread().getId()));
      deque.addFirst(event);
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

5. Create the `CleanerTask` class and specify that it extends the `Thread` class.

```
public class CleanerTask extends Thread {
```

6. Declare the queue that stores the events and implement the constructor of the class, which initializes this queue. In the constructor, mark this `Thread` as a daemon thread with the `setDaemon()` method.

```
  private Deque<Event> deque;
  public CleanerTask(Deque<Event> deque) {
    this.deque = deque;
    setDaemon(true);
  }
```

7. Implement the `run()` method. It has an infinite loop that gets the actual date and calls the `clean()` method.

```
  @Override
  public void run() {
    while (true) {
      Date date = new Date();
      clean(date);
    }
  }
```

8. Implement the `clean()` method. It gets the last event and, if it was created more than 10 seconds ago, it deletes it and checks the next event. If an event is deleted, it writes the message of the event and the new size of the queue, so you can see its evolution.

```
  private void clean(Date date) {
    long difference;
    boolean delete;

    if (deque.size()==0) {
      return;
    }
    delete=false;
    do {
      Event e = deque.getLast();
      difference = date.getTime() - e.getDate().getTime();
      if (difference > 10000) {
        System.out.printf("Cleaner: %s\n",e.getEvent());
        deque.removeLast();
        delete=true;
      }
    } while (difference > 10000);
    if (delete){
      System.out.printf("Cleaner: Size of the queue: %d\n",deque.
size());
    }
  }
```

9. Now, implement the main class. Create a class called `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

10. Create the queue to store the events using the `Deque` class.

```
    Deque<Event> deque=new ArrayDeque<Event>();
```

11. Create and start three `WriterTask` threads and one `CleanerTask`.

```
    WriterTask writer=new WriterTask(deque);
    for (int i=0; i<3; i++){
      Thread thread=new Thread(writer);
      thread.start();
    }
    CleanerTask cleaner=new CleanerTask(deque);
    cleaner.start();
```

12. Run the program and see the results.

## How it works...

If you analyze the output of one execution of the program, you can see how the queue begins to grow until it has 30 events and then, its size will vary between 27 and 30 events until the end of the execution.

The program starts with three `WriterTask` threads. Each `Thread` writes an event and sleeps for one second. After the first 10 seconds, we have 30 threads in the queue. During these 10 seconds, `CleanerTasks` has been executing while the three `WriterTask` threads were sleeping, but it hasn't deleted any event, because all of them were generated less than 10 seconds ago. During the rest of the execution, `CleanerTask` deletes three events every second and the three `WriterTask` threads write another three, so the size of the queue varies between 27 and 30 events.

You can play with the time until the `WriterTask` threads are sleeping. If you use a smaller value, you will see that `CleanerTask` has less CPU time and the size of the queue will increase because `CleanerTask` doesn't delete any event.

## There's more...

You only can call the `setDaemon()` method before you call the `start()` method. Once the thread is running, you can't modify its daemon status.

You can use the `isDaemon()` method to check if a thread is a daemon thread (the method returns `true`) or a user thread (the method returns `false`).

# Processing uncontrolled exceptions in a thread

There are two kinds of exceptions in Java:

- **Checked exceptions**: These exceptions must be specified in the `throws` clause of a method or caught inside them. For example, `IOException` or `ClassNotFoundException`.
- **Unchecked exceptions**: These exceptions don't have to be specified or caught. For example, `NumberFormatException`.

When a checked exception is thrown inside the `run()` method of a `Thread` object, we have to catch and treat them, because the `run()` method doesn't accept a `throws` clause. When an unchecked exception is thrown inside the `run()` method of a `Thread` object, the default behaviour is to write the stack trace in the console and exit the program.

Fortunately, Java provides us with a mechanism to catch and treat the unchecked exceptions thrown in a `Thread` object to avoid the program ending.

In this recipe, we will learn this mechanism using an example.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. First of all, we have to implement a class to treat the unchecked exceptions. This class must implement the `UncaughtExceptionHandler` interface and implement the `uncaughtException()` method declared in that interface. In our case, call this class `ExceptionHandler` and make the method to write information about `Exception` and `Thread` that threw it. Following is the code:

```
public class ExceptionHandler implements UncaughtExceptionHandler
{
  public void uncaughtException(Thread t, Throwable e) {
    System.out.printf("An exception has been captured\n");
    System.out.printf("Thread: %s\n",t.getId());
    System.out.printf("Exception: %s: %s\n",e.getClass().
getName(),e.getMessage());
    System.out.printf("Stack Trace: \n");
    e.printStackTrace(System.out);
    System.out.printf("Thread status: %s\n",t.getState());
  }
}
```

2. Now, implement a class that throws an unchecked exception. Call this class `Task`, specify that it implements the `Runnable` interface, implement the `run()` method, and force the exception, for example, try to convert a `string` value into an `int` value.

```
public class Task implements Runnable {
  @Override
  public void run() {
    int numero=Integer.parseInt("TTT");
  }
}
```

3. Now, implement the main class of the example. Implement a class called `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

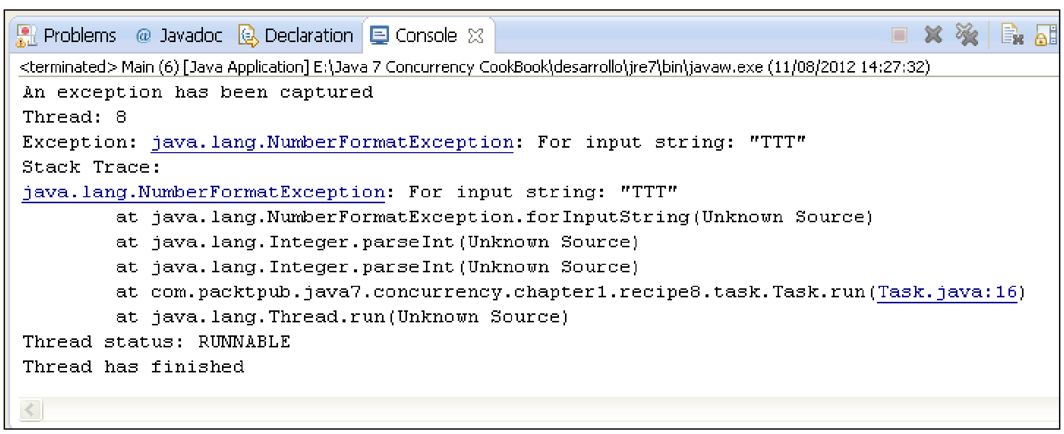
4. Create a `Task` object and `Thread` to run it. Set the unchecked exception handler using the `setUncaughtExceptionHandler()` method and start executing `Thread`.

```
    Task task=new Task();
    Thread thread=new Thread(task);
    thread.setUncaughtExceptionHandler(new ExceptionHandler());
    thread.start();
    }
}
```

5. Run the example and see the results.

## How it works...

In the following screenshot, you can see the results of the execution of the example. The exception is thrown and captured by the handler that writes the information in console about `Exception` and `Thread` that threw it. Refer to the following screenshot:




When an exception is thrown in a thread and is not caught (it has to be an unchecked exception), the JVM checks if the thread has an uncaught exception handler set by the corresponding method. If it has, the JVM invokes this method with the `Thread` object and `Exception` as arguments.

If the thread has not got an uncaught exception handler, the JVM prints the stack trace in the console and exits the program.

## There's more...

The `Thread` class has another method related to the process of uncaught exceptions. It's the static method `setDefaultUncaughtExceptionHandler()` that establishes an exception handler for all the `Thread` objects in the application.

When an uncaught exception is thrown in `Thread`, the JVM looks for three possible handlers for this exception.

First, it looks for the uncaught exception handler of the `Thread` objects as we learned in this recipe. If this handler doesn't exist, then the JVM looks for the uncaught exception handler for `ThreadGroup` of the `Thread` objects as was explained in the *Processing uncontrolled exceptions in a group of threads* recipe. If this method doesn't exist, the JVM looks for the default uncaught exception handler as we learned in this recipe.

If none of the handlers exits, the JVM writes the stack trace of the exception in the console and exits the program.

## See also

- The *Processing uncontrolled exceptions in a group of threads* recipe in *Chapter 1, Thread Management*

# Using local thread variables

One of the most critical aspects of a concurrent application is shared data. This has special importance in those objects that extend the `Thread` class or implement the `Runnable` interface.

If you create an object of a class that implements the `Runnable` interface and then start various `Thread` objects using the same `Runnable` object, all the threads share the same attributes. This means that, if you change an attribute in a thread, all the threads will be affected by this change.

Sometimes, you will be interested in having an attribute that won't be shared between all the threads that run the same object. The Java Concurrency API provides a clean mechanism called thread-local variables with a very good performance.

In this recipe, we will develop a program that has the problem exposed in the first paragraph and another program that solves this problem using the thread-local variables mechanism.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. First, we are going to implement a program that has the problem exposed previously. Create a class called `UnsafeTask` and specify that it implements the `Runnable` interface. Declare a `private` `java.util.Date` attribute.

```
public class UnsafeTask implements Runnable{
  private Date startDate;
```

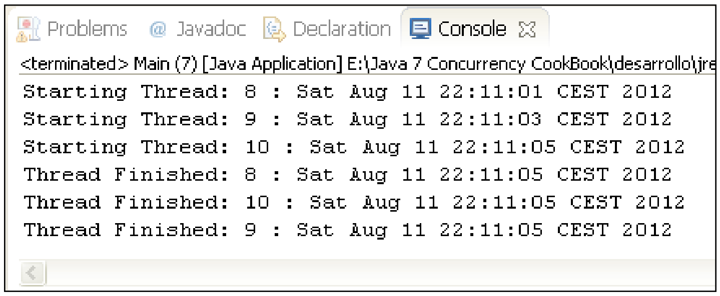
2. Implement the `run()` method of the `UnsafeTask` object. This method will initialize the `startDate` attribute, write its value to the console, sleep for a random period of time, and again write the value of the `startDate` attribute.

```
  @Override
  public void run() {
    startDate=new Date();
    System.out.printf("Starting Thread: %s : %s\n",Thread.
currentThread().getId(),startDate);
    try {
      TimeUnit.SECONDS.sleep( (int)Math.rint(Math.random()*10));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("Thread Finished: %s : %s\n",Thread.
currentThread().getId(),startDate);
  }
```

3. Now, let's implement the main class of this problematic application. Create a class called `Main` with a `main()` method. This method will create an object of the `UnsafeTask` class and start three threads using that object, sleeping for 2 seconds between each thread.

```
public class Core {
  public static void main(String[] args) {
    UnsafeTask task=new UnsafeTask();
    for (int i=0; i<10; i++){
      Thread thread=new Thread(task);
      thread.start();
      try {
        TimeUnit.SECONDS.sleep(2);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

4. In the following screenshot, you can see the results of this program's execution. Each `Thread` has a different start time but, when they finish, all have the same value in its `startDate` attribute.




5. As mentioned earlier, we are going to use the thread-local variables mechanism to solve this problem.

6. Create a class called `SafeTask` and specify that it implements the `Runnable` interface.

```
public class SafeTask implements Runnable {
```

7. Declare an object of the `ThreadLocal<Date>` class. This object will have an implicit implementation that includes the method `initialValue()`. This method will return the actual date.

```
  private static ThreadLocal<Date> startDate= new
ThreadLocal<Date>() {
    protected Date initialValue(){
      return new Date();
    }
  };
```

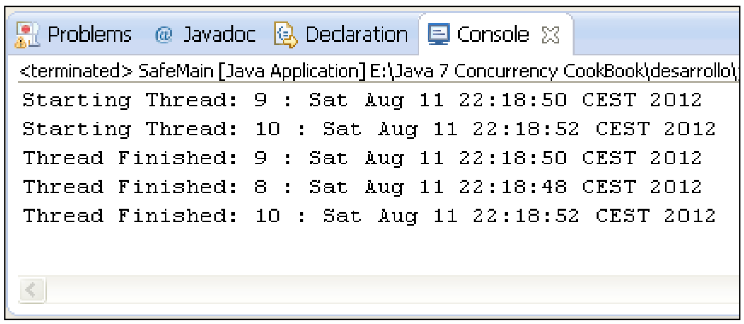
8. Implement the `run()` method. It has the same functionality as the `run()` method of `UnsafeClass`, but it changes the way to access to the `startDate` attribute.

```
  @Override
  public void run() {
    System.out.printf("Starting Thread: %s : %s\n",Thread.
currentThread().getId(),startDate.get());
    try {
      TimeUnit.SECONDS.sleep((int)Math.rint(Math.random()*10));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("Thread Finished: %s : %s\n",Thread.
currentThread().getId(),startDate.get());
  }
```

9. The main class of this example is the same as the unsafe example, changing the name of the `Runnable` class.
10. Run the example and analyze the difference.

## How it works...

In the following screenshot, you can see the results of the execution of the safe sample. Now, the three `Thread` objects have their own value of the `startDate` attribute. Refer to the following screenshot:




Thread-local variables store a value of an attribute for each `Thread` that uses one of these variables. You can read the value using the `get()` method and change the value using the `set()` method. The first time you access the value of a thread-local variable, if it has no value for the `Thread` object that it is calling, the thread-local variable calls the `initialValue()` method to assign a value for that `Thread` and returns the initial value.

## There's more...

The thread-local class also provides the `remove()` method that deletes the value stored in the thread-local variable for the thread that it's calling.

The Java Concurrency API includes the `InheritableThreadLocal` class that provides inheritance of values for threads created from a thread. If a thread A has a value in a thread-local variable and it creates another thread B, the thread B will have the same value as the thread A in the thread-local variable. You can override the `childValue()` method that is called to initialize the value of the child thread in the thread-local variable. It receives the value of the parent thread in the thread-local variable as a parameter.

# Grouping threads into a group

An interesting functionality offered by the concurrency API of Java is the ability to group the threads. This allows us to treat the threads of a group as a single unit and provides access to the `Thread` objects that belong to a group to do an operation with them. For example, you have some threads doing the same task and you want to control them, irrespective of how many threads are still running, the status of each one will interrupt all of them with a single call.

Java provides the `ThreadGroup` class to work with groups of threads. A `ThreadGroup` object can be formed by `Thread` objects and by another `ThreadGroup` object, generating a tree structure of threads.

In this recipe, we will learn to work with `ThreadGroup` objects developing a simple example. We will have 10 threads sleeping during a random period of time (simulating a search, for example) and, when one of them finishes, we are going to interrupt the rest.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. First, create a class called `Result`. It will store the name of `Thread` that finishes first. Declare a `private String` attribute called `name` and the methods to read and set the value.
2. Create a class called `SearchTask` and specify that it implements the `Runnable` interface.

   ```
   public class SearchTask implements Runnable {
   ```

3. Declare a `private` attribute of the `Result` class and implement the constructor of the class that initializes this attribute.

   ```
     private Result result;
     public SearchTask(Result result) {
       this.result=result;
     }
   ```

4. Implement the `run()` method. It will call the `doTask()` method and wait for it to finish or for a `InterruptedException` exception. The method will write messages to indicate the start, end, or interruption of this `Thread`.

   ```
     @Override
     public void run() {
   ```

```
      String name=Thread.currentThread().getName();
      System.out.printf("Thread %s: Start\n",name);
      try {
        doTask();
        result.setName(name);
      } catch (InterruptedException e) {
        System.out.printf("Thread %s: Interrupted\n",name);
        return;
      }
      System.out.printf("Thread %s: End\n",name);
    }
```

5. Implement the `doTask()` method. It will create a `Random` object to generate a random number and call the `sleep()` method with that random number.

```
  private void doTask() throws InterruptedException {
    Random random=new Random((new Date()).getTime());
    int value=(int)(random.nextDouble()*100);
    System.out.printf("Thread %s: %d\n",Thread.currentThread().
getName(),value);
    TimeUnit.SECONDS.sleep(value);
  }
```

6. Now, create the main class of the example by creating a class called `Main` and implement the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

7. First, create a `ThreadGroup` object and call them `Searcher`.

```
    ThreadGroup threadGroup = new ThreadGroup("Searcher");
```

8. Then, create a `SearchTask` object and a `Result` object.

```
    Result result=new Result();
    SearchTask searchTask=new SearchTask(result);
```

9. Now, create 10 `Thread` objects using the `SearchTask` object. When you call the constructor of the `Thread` class, pass it as the first argument of the `ThreadGroup` object.

```
    for (int i=0; i<5; i++) {
      Thread thread=new Thread(threadGroup, searchTask);
      thread.start();
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

10. Write information about the `ThreadGroup` object using the `list()` method.

```
    System.out.printf("Number of Threads: %d\n",threadGroup.
activeCount());
    System.out.printf("Information about the Thread Group\n");
    threadGroup.list();
```

11. Use the `activeCount()` and `enumerate()` methods to know how many `Thread` objects are associated with the `ThreadGroup` objects and get a list of them. We can use this method to get, for example, the state of each `Thread`.

```
    Thread[] threads=new Thread[threadGroup.activeCount()];
    threadGroup.enumerate(threads);
    for (int i=0; i<threadGroup.activeCount(); i++) {
      System.out.printf("Thread %s: %s\n",threads[i].
getName(),threads[i].getState());
    }
```

12. Call the method `waitFinish()`. We will implement this method later. It will wait until one of the threads of the `ThreadGroup` objects ends.

```
    waitFinish(threadGroup);
```

13. Interrupt the rest of the threads of the group using the `interrupt()` method.

```
    threadGroup.interrupt();
```

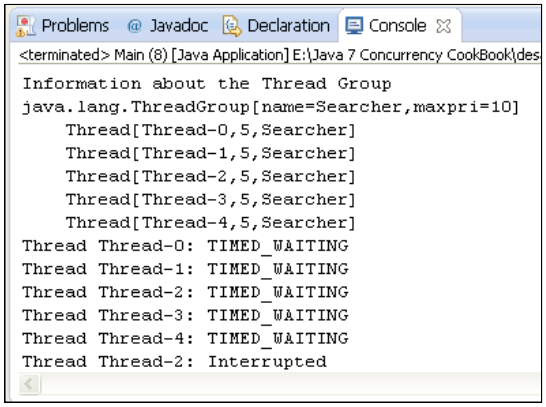
14. Implement the `waitFinish()` method. It will use the `activeCount()` method to control the end of one of the threads.

```
  private static void waitFinish(ThreadGroup threadGroup) {
    while (threadGroup.activeCount()>9) {
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

15. Run the example and see the results.

## How it works...

In the following screenshot, you can see the output of the `list()` method and the output generated when we write the status of each `Thread` object, as shown in the following screenshot:




The `ThreadGroup` class stores the `Thread` objects and the other `ThreadGroup` objects associated with it, so it can access all of their information (status, for example) and perform operations over all its members (interrupt, for example).

## There's more...

The `ThreadGroup` class has more methods. Check the API documentation to have a complete explanation of all of these methods.

# Processing uncontrolled exceptions in a group of threads

A very important aspect in every programming language is the mechanism that provides management of error situations in your application. Java language, as almost all modern programming languages, implements an exception-based mechanism to manage error situations. It provides a lot of classes to represent different errors. Those exceptions are thrown by the Java classes when an error situation is detected. You can also use those exceptions, or implement your own exceptions, to manage the errors produced in your classes.

Java also provides a mechanism to capture and process those exceptions. There are exceptions that must be captured or re-thrown using the `throws` clause of a method. These exceptions are called checked exceptions. There are exceptions that don't have to be specified or caught. These are the unchecked exceptions.

In the recipe, *Controlling the interruption of a Thread*, you learned how to use a generic method to process all the uncaught exceptions that are thrown in a `Thread` object.

Another possibility is to establish a method that captures all the uncaught exceptions thrown by any `Thread` of the `ThreadGroup` class.

In this recipe, we will learn to set this handler using an example.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. First, we have to extend the `ThreadGroup` class by creating a class called `MyThreadGroup` that extends from `ThreadGroup`. We have to declare a constructor with one parameter, because the `ThreadGroup` class doesn't have a constructor without it.

   ```
   public class MyThreadGroup extends ThreadGroup {
     public MyThreadGroup(String name) {
       super(name);
     }
   ```

2. Override the `uncaughtException()` method. This method is called when an exception is thrown in one of the threads of the `ThreadGroup` class. In this case, this method will write in the console information about the exception and `Thread` that throws it and interrupts the rest of the threads in the `ThreadGroup` class.

   ```
     @Override
     public void uncaughtException(Thread t, Throwable e) {
       System.out.printf("The thread %s has thrown an Exception\n",t.
   getId());
       e.printStackTrace(System.out);
       System.out.printf("Terminating the rest of the Threads\n");
       interrupt();
     }
   ```

3. Create a class called `Task` and specify that it implements the `Runnable` interface.

   ```
   public class Task implements Runnable {
   ```

4. Implement the `run()` method. In this case, we will provoke an `AritmethicException` exception. For this, we will divide 1000 between random numbers until the random generator generates a zero and the exception is thrown.

   ```
     @Override
     public void run() {
   ```

```
        int result;
        Random random=new Random(Thread.currentThread().getId());
        while (true) {
          result=1000/((int)(random.nextDouble()*1000));
          System.out.printf("%s : %f\n",Thread.currentThread().
    getId(),result);
          if (Thread.currentThread().isInterrupted()) {
            System.out.printf("%d : Interrupted\n",Thread.
    currentThread().getId());
            return;
          }
        }
      }
```

5. Now, we are going to implement the main class of the example by creating a class called `Main` and implement the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

6. Create an object of the `MyThreadGroup` class.

```
    MyThreadGroup threadGroup=new MyThreadGroup("MyThreadGroup");
```

7. Create an object of the `Task` class.

```
    Task task=new Task();
```

8. Create two `Thread` objects with this `Task` and start them.

```
    for (int i=0; i<2; i++){
      Thread t=new Thread(threadGroup,task);
      t.start();
    }
```

9. Run the example and see the results.

## How it works...

When you run the example, you will see how one of the `Thread` objects threw the exception and the other one was interrupted.

When an uncaught exception is thrown in `Thread`, the JVM looks for three possible handlers for this exception.

First, it looks for the uncaught exception handler of the thread, as was explained in the *Processing uncontrolled exceptions in a Thread* recipe. If this handler doesn't exist, then the JVM looks for the uncaught exception handler for the `ThreadGroup` class of the thread, as we learned in this recipe. If this method doesn't exist, the JVM looks for the default uncaught exception handler, as was explained in the *Processing uncontrolled exceptions in a Thread* recipe.

If none of the handlers exit, the JVM writes the stack trace of the exception in the console and exits the program.

## See also

- The *Processing uncontrolled exceptions in a thread* recipe in *Chapter 1, Thread Management*

# Creating threads through a factory

The factory pattern is one of the most used design patterns in the object-oriented programming world. It is a creational pattern and its objective is to develop an object whose mission will be creating other objects of one or several classes. Then, when we want to create an object of one of those classes, we use the factory instead of using the `new` operator.

With this factory, we centralize the creation of objects with some advantages:

- It's easy to change the class of the objects created or the way we create these objects.
- It's easy to limit the creation of objects for limited resources. For example, we can only have *n* objects of a type.
- It's easy to generate statistical data about the creation of the objects.

Java provides an interface, the `ThreadFactory` interface to implement a `Thread` object factory. Some advanced utilities of the Java concurrency API use thread factories to create threads.

In this recipe, we will learn how to implement a `ThreadFactory` interface to create `Thread` objects with a personalized name while we save statistics of the `Thread` objects created.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `MyThreadFactory` and specify that it implements the `ThreadFactory` interface.

```
public class MyThreadFactory implements ThreadFactory {
```

2. Declare three attributes: an integer number called `counter`, which we will use to store the number of the `Thread` object created, a `String` called `name` with the base name of every `Thread` created, and a `List` of `String` objects called `stats` to save statistical data about the `Thread` objects created. We also implement the constructor of the class that initializes these attributes.

```
  private int counter;
  private String name;
  private List<String> stats;

  public MyThreadFactory(String name){
    counter=0;
    this.name=name;
    stats=new ArrayList<String>();
  }
```

3. Implement the `newThread()` method. This method will receive a `Runnable` interface and returns a `Thread` object for this `Runnable` interface. In our case, we generate the name of the `Thread` object, create the new `Thread` object, and save the statistics.

```
  @Override
  public Thread newThread(Runnable r) {
    Thread t=new Thread(r,name+"-Thread_"+counter);
    counter++;
    stats.add(String.format("Created thread %d with name %s on 
%s\n",t.getId(),t.getName(),new Date()));
    return t;
  }
```

4. Implement the method `getStatistics()` that returns a `String` object with the statistical data of all the `Thread` objects created.

```
  public String getStats(){
    StringBuffer buffer=new StringBuffer();
    Iterator<String> it=stats.iterator();

    while (it.hasNext()) {
      buffer.append(it.next());
      buffer.append("\n");
```

```
    }

    return buffer.toString();
  }
```

5. Create a class called `Task` and specify that it implements the `Runnable` interface. For this example, these tasks are going to do nothing apart from sleeping for one second.

```
public class Task implements Runnable {
  @Override
  public void run() {
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

6. Create the main class of the example. Create a class called `Main` and implement the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

7. Create a `MyThreadFactory` object and a `Task` object.

```
    MyThreadFactory factory=new MyThreadFactory("MyThreadFactory")
;
    Task task=new Task();
```

8. Create 10 `Thread` objects using the `MyThreadFactory` object and start them.

```
    Thread thread;
    System.out.printf("Starting the Threads\n");
    for (int i=0; i<10; i++){
      thread=factory.newThread(task);
      thread.start();
    }
```

9. Write in the console the statistics of the thread factory.

```
    System.out.printf("Factory stats:\n");
    System.out.printf("%s\n",factory.getStats());
```

10. Run the example and see the results.

## How it works...

The `ThreadFactory` interface has only one method called `newThread`. It receives a `Runnable` object as a parameter and returns a `Thread` object. When you implement a `ThreadFactory` interface, you have to implement that interface and override this method. Most basic `ThreadFactory`, has only one line.

```
return new Thread(r);
```

You can improve this implementation by adding some variants by:

- Creating personalized threads, as in the example, using a special format for the name or even creating our own `thread` class that inherits the Java `Thread` class
- Saving thread creation statistics, as shown in the previous example
- Limiting the number of threads created
- Validating the creation of the threads
- And anything more you can imagine

The use of the factory design pattern is a good programming practice but, if you implement a `ThreadFactory` interface to centralize the creation of threads, you have to review the code to guarantee that all threads are created using that factory.

## See also

- The *Implementing the ThreadFactory interface to generate custom threads* recipe in *Chapter 7*, *Customizing Concurrency Classes*
- The *Using our ThreadFactory in an Executor object* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# 2

# Basic Thread Synchronization

In this chapter, we will cover:

- Synchronizing a method
- Arranging independent attributes in synchronized classes
- Using conditions in synchronized code
- Synchronizing a block of code with a Lock
- Synchronizing data access with read/write locks
- Modifying Lock fairness
- Using multiple conditions in a Lock

## Introduction

One of the most common situations in concurrent programming occurs when more than one execution thread shares a resource. In a concurrent application, it is normal that multiple threads read or write the same data or have access to the same file or database connection. These shared resources can provoke error situations or data inconsistency and we have to implement mechanisms to avoid these errors.

The solution for these problems comes with the concept of **critical section**. A critical section is a block of code that accesses a shared resource and can't be executed by more than one thread at the same time.

To help programmers to implement critical sections, Java (and almost all programming languages) offers **synchronization** mechanisms. When a thread wants access to a critical section, it uses one of those synchronization mechanisms to find out if there is any other thread executing the critical section. If not, the thread enters the critical section. Otherwise, the thread is suspended by the synchronization mechanism until the thread that is executing the critical section ends it. When more than one thread is waiting for a thread to finish the execution of a critical section, the JVM chooses one of them, and the rest wait for their turn.

This chapter presents a number of recipes that teaches how to use the two basic synchronization mechanisms offered by the Java language:

- The keyword `synchronized`
- The `Lock` interface and its implementations

# Synchronizing a method

In this recipe, we will learn how to use one of the most basic methods for synchronization in Java, that is, the use of the `synchronized` keyword to control the concurrent access to a method. Only one execution thread will access one of the methods of an object declared with the `synchronized` keyword. If another thread tries to access any method declared with the `synchronized` keyword of the same object, it will be suspended until the first thread finishes the execution of the method.

In other words, every method declared with the `synchronized` keyword is a critical section and Java only allows the execution of one of the critical sections of an object.

Static methods have a different behavior. Only one execution thread will access one of the static methods declared with the `synchronized` keyword, but another thread can access other non-static methods of an object of that class. You have to be very careful with this point, because two threads can access two different `synchronized` methods if one is static and the other one is not. If both methods change the same data, you can have data inconsistency errors.

To learn this concept, we will implement an example with two threads accessing a common object. We will have a bank account and two threads; one that transfers money to the account and another one that withdraws money from the account. Without synchronization methods, we could have incorrect results. Synchronization mechanisms ensures that the final balance of the account will be correct.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `Account` that will model our bank account. It has only one `double` attribute, named `balance`.

   ```
   public class Account {
         private double balance;
   ```

2. Implement the `setBalance()` and `getBalance()` methods to write and read the value of the attribute.

   ```
     public double getBalance() {
       return balance;
     }

     public void setBalance(double balance) {
       this.balance = balance;
     }
   ```

3. Implement a method called `addAmount()` that increments the value of the balance in a certain amount that is passed to the method. Only one thread should change the value of the balance, so use the `synchronized` keyword to convert this method into a critical section.

   ```
     public synchronized void addAmount(double amount) {
       double tmp=balance;
       try {
         Thread.sleep(10);
       } catch (InterruptedException e) {
         e.printStackTrace();
       }
       tmp+=amount;
       balance=tmp;
     }
   ```

4. Implement a method called `subtractAmount()` that decrements the value of the balance in a certain amount that is passed to the method. Only one thread should change the value of the balance, so use the `synchronized` keyword to convert this method into a critical section.

   ```
     public synchronized void subtractAmount(double amount) {
       double tmp=balance;
       try {
         Thread.sleep(10);
       } catch (InterruptedException e) {
         e.printStackTrace();
   ```

```
    }
    tmp-=amount;
    balance=tmp;
  }
```

5. Implement a class that simulates an ATM. It will use the `subtractAmount()` method to decrement the balance of an account. This class must implement the `Runnable` interface to be executed as a thread.

```
public class Bank implements Runnable {
```

6. Add an `Account` object to this class. Implement the constructor of the class that initializes that `Account` object.

```
  private Account account;

  public Bank(Account account) {
    this.account=account;
  }
```

7. Implement the `run()` method. It makes `100` calls to the `subtractAmount()` method of an account to reduce the balance.

```
  @Override
   public void run() {
    for (int i=0; i<100; i++){
      account.sustractAmount(1000);
    }
  }
```

8. Implement a class that simulates a company and uses the `addAmount()` method of the `Account` class to increment the balance of the account. This class must implement the `Runnable` interface to be executed as a thread.

```
public class Company implements Runnable {
```

9. Add an `Account` object to this class. Implement the constructor of the class that initializes that account object.

```
  private Account account;

  public Company(Account account) {
    this.account=account;
  }
```

10. Implement the `run()` method . It makes `100` calls to the `addAmount()` method of an account to increment the balance.

```
  @Override
   public void run() {
    for (int i=0; i<100; i++){
```

```
      account.addAmount(1000);
    }
  }
```

11. Implement the main class of the application by creating a class named `Main` that contains the `main()` method.

```
public class Main {

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

12. Create an `Account` object and initialize its balance to `1000`.

```
    Account  account=new Account();
    account.setBalance(1000);
```

13. Create a `Company` object and `Thread` to run it.

```
    Company  company=new Company(account);
    Thread companyThread=new Thread(company);
```

14. Create a `Bank` object and `Thread` to run it.

```
    Bank bank=new Bank(account);
    Thread bankThread=new Thread(bank);
```

15. Write the initial balance to the console.

```
    System.out.printf("Account : Initial Balance: %f\n",account.
getBalance());
Start the threads.
    companyThread.start();
    bankThread.start();
```

16. Wait for the finalization of the two threads using the `join()` method and print in the console the final balance of the account.

```
    try {
      companyThread.join();
      bankThread.join();
      System.out.printf("Account : Final Balance: %f\n",account.
getBalance());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

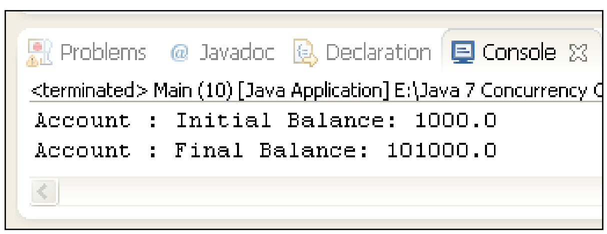
## How it works...

In this recipe, you have developed an application that increments and decrements the balance of a class that simulates a bank account. The program makes `100` calls to the `addAmount()` method that increments the balance by `1000` in each call and `100` calls to the `subtractAmount()` method that decrements the balance by `1000` in each call. You should expect the final and initial balances to be equal.

You have tried to force an error situation using a variable named `tmp` to store the value of the account's balance, so you read the account's balance, you increment the value of the temporal variable, and then you establish the value of the account's balance again. Additionally, you have introduced a little delay using the `sleep()` method of the `Thread` class to put the thread that is executing the method to sleep for 10 milliseconds, so if another thread executes that method, it can modify the account's balance provoking an error. It's the `synchronized` keyword mechanism that avoids those errors.

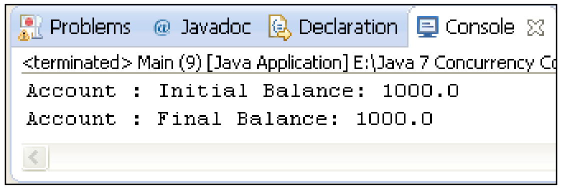
If you want to see the problems of concurrent access to shared data, delete the `synchronized` keyword of the `addAmount()` and `subtractAmount()` methods and run the program. Without the `synchronized` keyword, while a thread is sleeping after reading the value of the account's balance, another method will read the account's balance, so both the methods will modify the same balance and one of the operations won't be reflected in the final result.

As you can see in the following screenshot, you can obtain inconsistent results:




If you run the program often, you will obtain different results. The order of execution of the threads is not guaranteed by the JVM. So every time you execute them, the threads will read and modify the account's balance in a different order, so the final result will be different.

Now, add the `synchronize` keyword as you learned before and run the program again. As you can see in the following screenshot, now you obtain the expected result. If you run the program often, you will obtain the same result. Refer to the following screenshot:




Using the `synchronized` keyword, we guarantee correct access to shared data in concurrent applications.

As we mentioned in the introduction of this recipe, only a thread can access the methods of an object that use the `synchronized` keyword in their declaration. If a thread (A) is executing a `synchronized` method and another thread (B) wants to execute other `synchronized` methods of the same object, it will be blocked until the thread (A) ends. But if threadB has access to different objects of the same class, none of them will be blocked.

## There's more...

The `synchronized` keyword penalizes the performance of the application, so you must only use it on methods that modify shared data in a concurrent environment. If you have multiple threads calling a `synchronized` method, only one will execute them at a time while the others will be waiting. If the operation doesn't use the `synchronized` keyword, all the threads can execute the operation at the same time, reducing the total execution time. If you know that a method will not be called by more than one thread, don't use the `synchronized` keyword.

You can use recursive calls with `synchronized` methods. As the thread has access to the `synchronized` methods of an object, you can call other `synchronized` methods of that object, including the method that is executing. It won't have to get access to the `synchronized` methods again.

We can use the `synchronized` keyword to protect the access to a block of code instead of an entire method. We should use the `synchronized` keyword in this way to protect the access to the shared data, leaving the rest of operations out of this block, obtaining a better performance of the application. The objective is to have the critical section (the block of code that can be accessed only by one thread at a time) be as short as possible. We have used the `synchronized` keyword to protect the access to the instruction that updates the number of persons in the building, leaving out the long operations of this block that don't use the shared data. When you use the `synchronized` keyword in this way, you must pass an object reference as a parameter. Only one thread can access the `synchronized` code (blocks or methods) of that object. Normally, we will use the `this` keyword to reference the object that is executing the method.

```
synchronized (this) {
  // Java code
}
```

# Arranging independent attributes in synchronized classes

When you use the `synchronized` keyword to protect a block of code, you must pass an object reference as a parameter. Normally, you will use the `this` keyword to reference the object that executes the method, but you can use other object references. Normally, these objects will be created exclusively with this purpose. For example, if you have two independent attributes in a class shared by multiple threads, you must synchronize the access to each variable, but there is no problem if there is one thread accessing one of the attributes and another thread accessing the other at the same time.

In this recipe, you will learn how to resolve this situation's programming with an example that simulates a cinema with two screens and two ticket offices. When a ticket office sells tickets, they are for one of the two cinemas, but not for both, so the numbers of free seats in each cinema are independent attributes.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class called `Cinema` and add to it two `long` attributes named `vacanciesCinema1` and `vacanciesCinema2`.

```
public class Cinema {

  private long vacanciesCinema1;
  private long vacanciesCinema2;
```

2. Add to the `Cinema` class two additional `Object` attributes named `controlCinema1` and `controlCinema2`.

```
  private final Object controlCinema1, controlCinema2;
```

3. Implement the constructor of the `Cinema` class that initializes all the attributes of the class.

```
  public Cinema(){
    controlCinema1=new Object();
    controlCinema2=new Object();
    vacanciesCinema1=20;
    vacanciesCinema2=20;
  }
```

4. Implement the `sellTickets1()` method that is called when some tickets for the first cinema are sold. It uses the `controlCinema1` object to control the access to the `synchronized` block of code.

```
public boolean sellTickets1 (int number) {
  synchronized (controlCinema1) {
    if (number<vacanciesCinema1) {
      vacanciesCinema1-=number;
      return true;
    } else {
      return false;
    }
  }
}
```

5. Implement the `sellTickets2()` method that is called when some tickets for the second cinema are sold. It uses the `controlCinema2` object to control the access to the `synchronized` block of code.

```
public boolean sellTickets2 (int number){
  synchronized (controlCinema2) {
    if (number<vacanciesCinema2) {
      vacanciesCinema2-=number;
      return true;
    } else {
      return false;
    }
  }
}
```

6. Implement the `returnTickets1()` method that is called when some tickets for the first cinema are returned. It uses the `controlCinema1` object to control the access to the `synchronized` block of code.

```
public boolean returnTickets1 (int number) {
  synchronized (controlCinema1) {
    vacanciesCinema1+=number;
    return true;
  }
}
```

7. Implement the `returnTickets2()` method that is called when some tickets for the second cinema are returned. It uses the `controlCinema2` object to control the access to the `synchronized` block of code.

```
public boolean returnTickets2 (int number) {
  synchronized (controlCinema2) {
    vacanciesCinema2+=number;
```

```
      return true;
    }
  }
```

8. Implement another two methods that return the number of vacancies in each cinema.

```
  public long getVacanciesCinema1() {
    return vacanciesCinema1;
  }

  public long getVacanciesCinema2() {
    return vacanciesCinema2;
  }
```

9. Implement the class `TicketOffice1` and specify that it implements the `Runnable` interface.

```
public class TicketOffice1 implements Runnable {
```

10. Declare a `Cinema` object and implement the constructor of the class that initializes that object.

```
  private Cinema cinema;

  public TicketOffice1 (Cinema cinema) {
    this.cinema=cinema;
  }
```

11. Implement the `run()` method that simulates some operations over the two cinemas.

```
  @Override
   public void run() {
    cinema.sellTickets1(3);
    cinema.sellTickets1(2);
    cinema.sellTickets2(2);
    cinema.returnTickets1(3);
    cinema.sellTickets1(5);
    cinema.sellTickets2(2);
    cinema.sellTickets2(2);
    cinema.sellTickets2(2);
  }
```

12. Implement the class `TicketOffice2` and specify that it implements the `Runnable` interface.

```
public class TicketOffice2 implements Runnable {
```

13. Declare a `Cinema` object and implement the constructor of the class that initializes that object.

```
private Cinema cinema;

public TicketOffice2(Cinema cinema){
  this.cinema=cinema;
}
```

14. Implement the `run()` method that simulates some operations over the two cinemas.

```
@Override
public void run() {
  cinema.sellTickets2(2);
  cinema.sellTickets2(4);
  cinema.sellTickets1(2);
  cinema.sellTickets1(1);
  cinema.returnTickets2(2);
  cinema.sellTickets1(3);
  cinema.sellTickets2(2);
  cinema.sellTickets1(2);
}
```

15. Implement the main class of the example by creating a class called `Main` and add to it the `main()` method.

```
public class Main {

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

16. Declare and create a `Cinema` object.

```
    Cinema cinema=new Cinema();
```

17. Create a `TicketOffice1` object and `Thread` to execute it.

```
    TicketOffice1 ticketOffice1=new TicketOffice1(cinema);
    Thread thread1=new Thread(ticketOffice1,"TicketOffice1");
```

18. Create a `TicketOffice2` object and `Thread` to execute it.

```
    TicketOffice2 ticketOffice2=new TicketOffice2(cinema);
    Thread thread2=new Thread(ticketOffice2,"TicketOffice2");
```

19. Start both threads.

```
    thread1.start();
    thread2.start();
```

20. Wait for the completion of the threads.

```
try {
  thread1.join();
  thread2.join();
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

21. Write to the console the vacancies of the two cinemas.

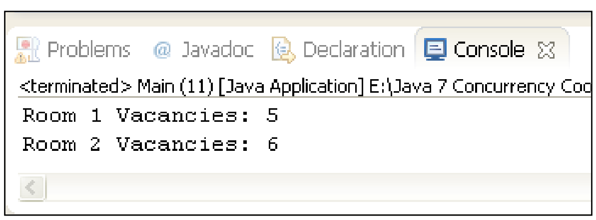
```
    System.out.printf("Room 1 Vacancies: %d\n",cinema.
getVacanciesCinema1());
    System.out.printf("Room 2 Vacancies: %d\n",cinema.
getVacanciesCinema2());
```

## How it works...

When you use the `synchronized` keyword to protect a block of code, you use an object as a parameter. JVM guarantees that only one thread can have access to all the blocks of code protected with that object (note that we always talk about objects, not about classes).

> In this example, we have an object that controls access to the `vacanciesCinema1` attribute, so only one thread can modify this attribute each time, and another object controls access to the `vacanciesCinema2` attribute, so only one thread can modify this attribute each time. But there may be two threads running simultaneously, one modifying the `vacancesCinema1` attribute and the other one modifying the `vacanciesCinema2` attribute.

When you run this example, you can see how the final result is always the expected number of vacancies for each cinema. In the following screenshot, you can see the results of an execution of the application:




## There's more...

There are other important uses of the `synchronize` keyword. See the *See also* section for other recipes that explain the use of this keyword.

## See also

- The *Using conditions in synchronized code* recipe in *Chapter 2*, *Basic Thread Synchronization*

# Using conditions in synchronized code

A classic problem in concurrent programming is the **producer-consumer** problem. We have a data buffer, one or more producers of data that save it in the buffer and one or more consumers of data that take it from the buffer.

As the buffer is a shared data structure, we have to control the access to it using a synchronization mechanism such as the `synchronized` keyword, but we have more limitations. A producer can't save data in the buffer if it's full and the consumer can't take data from the buffer if it's empty.

For these types of situations, Java provides the `wait()`, `notify()`, and `notifyAll()` methods implemented in the `Object` class. A thread can call the `wait()` method inside a `synchronized` block of code. If it calls the `wait()` method outside a `synchronized` block of code, the JVM throws an `IllegalMonitorStateException` exception. When the thread calls the `wait()` method, the JVM puts the thread to sleep and releases the object that controls the `synchronized` block of code that it's executing and allows the other threads to execute other blocks of `synchronized` code protected by that object. To wake up the thread, you must call the `notify()` or `notifyAll()` method inside a block of code protected by the same object.

In this recipe, you will learn how to implement the producer-consumer problem using the `synchronized` keyword and the `wait()`, `notify()`, and `notifyAll()` methods.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `EventStorage`. It has two attributes: an `int` attribute called `maxSize` and a `LinkedList<Date>` attribute called `storage`.

```
public class EventStorage {

  private int maxSize;
  private List<Date> storage;
```

2. Implement the constructor of the class that initializes the attributes of the class.

```
public EventStorage(){
  maxSize=10;
  storage=new LinkedList<>();
}
```

3. Implement the synchronized method set() to store an event in the storage. First, check if the storage is full or not. If it's full, it calls the wait() method until the storage has empty space. At the end of the method, we call the notifyAll() method to wake up all the threads that are sleeping in the wait() method.

```
public synchronized void set(){
    while (storage.size()==maxSize){
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    storage.offer(new Date());
    System.out.printf("Set: %d",storage.size());
    notifyAll();
}
```

4. Implement the synchronized method get() to get an event for the storage. First, check if the storage has events or not. If it has no events, it calls the wait() method until the storage has some events. At the end of the method, we call the notifyAll() method to wake up all the threads that are sleeping in the wait() method.

```
public synchronized void get(){
    while (storage.size()==0){
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
     System.out.printf("Get: %d: %s",storage.
size(),((LinkedList<?>)storage).poll());
    notifyAll();
}
```

5. Create a class named Producer and specify that it implements the Runnable interface. It will implement the producer of the example.

```
public class Producer implements Runnable {
```

6. Declare an `EventStore` object and implement the constructor of the class that initializes that object.

```
private EventStorage storage;

public Producer(EventStorage storage){
  this.storage=storage;
}
```

7. Implement the `run()` method that calls `100` times the `set()` method of the `EventStorage` object.

```
 @Override
public void run() {
  for (int i=0; i<100; i++){
    storage.set();
  }
}
```

8. Create a class named `Consumer` and specify that it implements the `Runnable` interface. It will implement the consumer for the example.

```
public class Consumer implements Runnable {
```

9. Declare an `EventStorage` object and implement the constructor of the class that initializes that object.

```
private EventStorage storage;

public Consumer(EventStorage storage){
  this.storage=storage;
}
```

10. Implement the `run()` method. It calls `100` times the `get()` method of the `EventStorage` object.

```
@Override
 public void run() {
  for (int i=0; i<100; i++){
    storage.get();
  }
}
```

11. Create the main class of the example by implementing a class named `Main` and add to it the `main()` method.

```
public class Main {

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

12. Create an `EventStorage` object.

    ```
    EventStorage storage=new EventStorage();
    ```

13. Create a `Producer` object and `Thread` to run it.

    ```
    Producer producer=new Producer(storage);
    Thread thread1=new Thread(producer);
    ```

14. Create a `Consumer` object and `Thread` to run it.

    ```
    Consumer consumer=new Consumer(storage);
    Thread thread2=new Thread(consumer);
    ```

15. Start both threads.

    ```
    thread2.start();
    thread1.start();
    ```

## How it works...

The key to this example is the `set()` and `get()` methods of the `EventStorage` class. First of all, the `set()` method checks if there is free space in the storage attribute. If it's full, it calls the `wait()` method to wait for free space. When the other thread calls the `notifyAll()` method, the thread wakes up and checks the condition again. The `notifyAll()` method doesn't guarantee that the thread will wake up. This process is repeated until there is free space in the storage and it can generate a new event and store it.

The behavior of the `get()` method is similar. First, it checks if there are events on the storage. If the `EventStorage` class is empty, it calls the `wait()` method to wait for events. Where the other thread calls the `notifyAll()` method, the thread wakes up and checks the condition again until there are some events in the storage.

> You have to keep checking the conditions and calling the `wait()` method in a `while` loop. You can't continue until the condition is `true`.

If you run this example, you will see how producer and consumer are setting and getting the events, but the storage never has more than 10 events.

## There's more...

There are other important uses of the `synchronized` keyword. See the *See also* section for other recipes that explain the use of this keyword.

## See also

- The *Arranging independent attributes in synchronized classes* recipe in *Chapter 2, Basic Thread Synchronization*

# Synchronizing a block of code with a Lock

Java provides another mechanism for the synchronization of blocks of code. It's a more powerful and flexible mechanism than the `synchronized` keyword. It's based on the `Lock` interface and classes that implement it (as `ReentrantLock`). This mechanism presents some advantages, which are as follows:

- It allows the structuring of synchronized blocks in a more flexible way. With the `synchronized` keyword, you have to get and free the control over a synchronized block of code in a structured way. The `Lock` interfaces allow you to get more complex structures to implement your critical section.
- The `Lock` interfaces provide additional functionalities over the `synchronized` keyword. One of the new functionalities is implemented by the `tryLock()` method. This method tries to get the control of the lock and if it can't, because it's used by other thread, it returns the lock. With the `synchronized` keyword, when a thread (A) tries to execute a synchronized block of code, if there is another thread (B) executing it, the thread (A) is suspended until the thread (B) finishes the execution of the synchronized block. With locks, you can execute the `tryLock()` method. This method returns a `Boolean` value indicating if there is another thread running the code protected by this lock.
- The `Lock` interfaces allow a separation of read and write operations having multiple readers and only one modifier.
- The `Lock` interfaces offer better performance than the `synchronized` keyword.

In this recipe, you will learn how to use locks to synchronize a block of code and create a critical section using the `Lock` interface and the `ReentrantLock` class that implements it, implementing a program that simulates a print queue.

## Getting Ready...

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `PrintQueue` that will implement the print queue.

```
public class PrintQueue {
```

2. Declare a `Lock` object and initialize it with a new object of the `ReentrantLock` class.

```
  private final Lock queueLock=new ReentrantLock();
```

3. Implement the `printJob()` method. It will receive `Object` as a parameter and it will not return any value.

```
  public void printJob(Object document){
```

4. Inside the `printJob()` method, get the control of the `Lock` object calling the `lock()` method.

```
    queueLock.lock();
```

5. Then, include the following code to simulate the printing of a document:

```
    try {
      Long duration=(long)(Math.random()*10000);
      System.out.println(Thread.currentThread().getName()+ ":
PrintQueue: Printing a Job during "+(duration/1000)+
" seconds");
      Thread.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

6. Finally, free the control of the `Lock` object with the `unlock()` method.

```
finally {
      queueLock.unlock();
    }
```

7. Create a class named `Job` and specify that it implements the `Runnable` interface.

```
public class Job implements Runnable {
```

8. Declare an object of the `PrintQueue` class and implement the constructor of the class that initializes that object.

```
  private PrintQueue printQueue;

  public Job(PrintQueue printQueue){
    this.printQueue=printQueue;
  }
```

9. Implement the `run()` method. It uses the `PrintQueue` object to send a job to print.

```
  @Override
  public void run() {
    System.out.printf("%s: Going to print a document\n", Thread.
currentThread().getName());
    printQueue.printJob(new Object());
    System.out.printf("%s: The document has been printed\n",
Thread.currentThread().getName());
  }
```

10. Create the main class of the application by implementing a class named `Main` and add the `main()` method to it.

```
public class Main {

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

11. Create a shared `PrintQueue` object.

```
    PrintQueue printQueue=new PrintQueue();
```

12. Create 10 `Job` objects and 10 threads to run them.

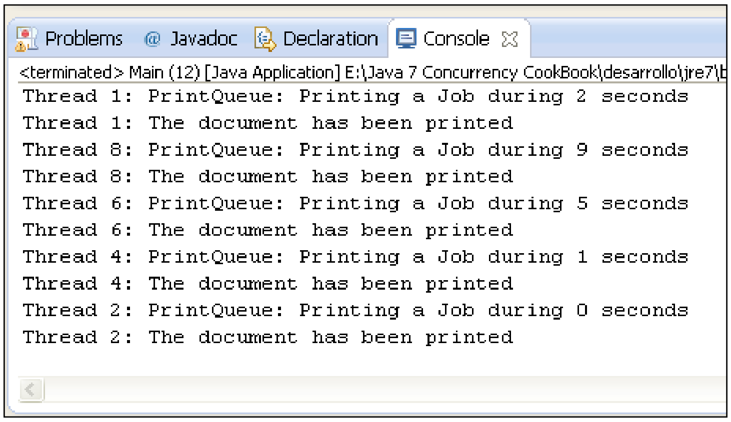
```
    Thread thread[]=new Thread[10];
    for (int i=0; i<10; i++){
      thread[i]=new Thread(new Job(printQueue),"Thread "+ i);
    }
```

13. Start the 10 threads.

```
    for (int i=0; i<10; i++){
      thread[i].start();
    }
```

## How it works...

In the following screenshot, you can see a part of the output of one execution, of this example:




The key to the example is in the `printJob()` method of the `PrintQueue` class. When we want to implement a critical section using locks and guarantee that only one execution thread runs a block of code, we have to create a `ReentrantLock` object. At the beginning of the critical section, we have to get the control of the lock using the `lock()` method. When a thread (A) calls this method, if no other thread has the control of the lock, the method gives the thread (A) the control of the lock and returns immediately to permit the execution of the critical section to this thread. Otherwise, if there is another thread (B) executing the critical section controlled by this lock, the `lock()` method puts the thread (A) to sleep until the thread (B) finishes the execution of the critical section.

At the end of the critical section, we have to use the `unlock()` method to free the control of the lock and allow the other threads to run this critical section. If you don't call the `unlock()` method at the end of the critical section, the other threads that are waiting for that block will be waiting forever, causing a deadlock situation. If you use try-catch blocks in your critical section, don't forget to put the sentence containing the `unlock()` method inside the `finally` section.

## There's more...

The `Lock` interface (and the `ReentrantLock` class) includes another method to get the control of the lock. It's the `tryLock()` method. The biggest difference with the `lock()` method is that this method, if the thread that uses it can't get the control of the `Lock` interface, returns immediately and doesn't put the thread to sleep. This method returns a `boolean` value, `true` if the thread gets the control of the lock, and `false` if not.

> Take into consideration that it is the responsibility of the programmer to take into account the result of this method and act accordingly. If the method returns the `false` value, it's expected that your program doesn't execute the critical section. If it does, you probably will have wrong results in your application.

The `ReentrantLock` class also allows the use of recursive calls. When a thread has the control of a lock and makes a recursive call, it continues with the control of the lock, so the calling to the `lock()` method will return immediately and the thread will continue with the execution of the recursive call. Moreover, we can also call other methods.

### More Info

You have to be very careful with the use of `Locks` to avoid **deadlocks**. This situation occurs when two or more threads are blocked waiting for locks that never will be unlocked. For example, a thread (A) locks a Lock (X) and a thread (B) locks a Lock (Y). If now, the thread (A) tries to lock the Lock (Y) and the thread (B) simultaneously tries to lock the Lock (X), both threads will be blocked indefinitely, because they are waiting for locks that will never be liberated. Note that the problem occurs, because both threads try to get the locks in the opposite order. The *Appendix*, *Concurrent programming design*, explains some good tips to design concurrent applications adequately and avoid these deadlocks problems.

## See also

- The *Synchronizing a method* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Using multiple conditions in a Lock* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Monitoring a Lock* interface recipe in *Chapter 8*, *Testing Concurrent Applications*

# Synchronizing data access with read/write locks

One of the most significant improvements offered by locks is the `ReadWriteLock` interface and the `ReentrantReadWriteLock` class, the unique one that implements it. This class has two locks, one for read operations and one for write operations. There can be more than one thread using read operations simultaneously, but only one thread can be using write operations. When a thread is doing a write operation, there can't be any thread doing read operations.

In this recipe, you will learn how to use a `ReadWriteLock` interface implementing a program that uses it to control the access to an object that stores the prices of two products.

## Getting Ready...

You should read the *Synchronizing a block of code with a Lock* recipe for a better understanding of this recipe.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `PricesInfo` that stores information about the prices of two products.

   ```
   public class PricesInfo {
   ```

2. Declare two `double` attributes named `price1` and `price2`.

   ```
     private double price1;
     private double price2;
   ```

3. Declare a `ReadWriteLock` object called `lock`.

   ```
     private ReadWriteLock lock;
   ```

4. Implement the constructor of the class that initializes the three attributes. For the `lock` attribute, we create a new `ReentrantReadWriteLock` object.

   ```
     public PricesInfo(){
       price1=1.0;
       price2=2.0;
       lock=new ReentrantReadWriteLock();
     }
   ```

5. Implement the `getPrice1()` method that returns the value of the `price1` attribute. It uses the read lock to control the access to the value of this attribute.

   ```
     public double getPrice1() {
       lock.readLock().lock();
       double value=price1;
       lock.readLock().unlock();
       return value;
     }
   ```

6. Implement the `getPrice2()` method that returns the value of the `price2` attribute. It uses the read lock to control the access to the value of this attribute.

   ```
     public double getPrice2() {
       lock.readLock().lock();
       double value=price2;
       lock.readLock().unlock();
       return value;
     }
   ```

7. Implement the `setPrices()` method that establishes the values of the two attributes. It uses the write lock to control access to them.

```
  public void setPrices(double price1, double price2) {
    lock.writeLock().lock();
    this.price1=price1;
    this.price2=price2;
    lock.writeLock().unlock();
  }
```

8. Create a class named `Reader` and specify that it implements the `Runnable` interface. This class implements a reader of the values of the `PricesInfo` class attributes.

```
public class Reader implements Runnable {
```

9. Declare a `PricesInfo` object and implement the constructor of the class that initializes that object.

```
  private PricesInfo pricesInfo;

  public Reader (PricesInfo pricesInfo){
    this.pricesInfo=pricesInfo;
  }
```

10. Implement the `run()` method for this class. It reads 10 times the value of the two prices.

```
  @Override
  public void run() {
    for (int i=0; i<10; i++){
      System.out.printf("%s: Price 1: %f\n", Thread.
currentThread().getName(),pricesInfo.getPrice1());
      System.out.printf("%s: Price 2: %f\n", Thread.
currentThread().getName(),pricesInfo.getPrice2());
    }
  }
```

11. Create a class named `Writer` and specify that it implements the `Runnable` interface. This class implements a modifier of the values of the `PricesInfo` class attributes.

```
public class Writer implements Runnable {
```

12. Declare a `PricesInfo` object and implement the constructor of the class that initializes that object.

```
  private PricesInfo pricesInfo;

  public Writer(PricesInfo pricesInfo){
    this.pricesInfo=pricesInfo;
  }
```

13. Implement the `run()` method. It modifies three times the value of the two prices that are sleeping for two seconds between modifications.

```
  @Override
  public void run() {
    for (int i=0; i<3; i++) {
      System.out.printf("Writer: Attempt to modify the
prices.\n");
      pricesInfo.setPrices(Math.random()*10, Math.random()*8);
      System.out.printf("Writer: Prices have been modified.\n");
      try {
        Thread.sleep(2);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

14. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

15. Create a `PricesInfo` object.

```
    PricesInfo pricesInfo=new PricesInfo();
```

16. Create five `Reader` objects and five `Threads` to execute them.

```
    Reader readers[]=new Reader[5];
    Thread threadsReader[]=new Thread[5];

    for (int i=0; i<5; i++){
      readers[i]=new Reader(pricesInfo);
      threadsReader[i]=new Thread(readers[i]);
    }
```

17. Create a `Writer` object and `Thread` to execute it.

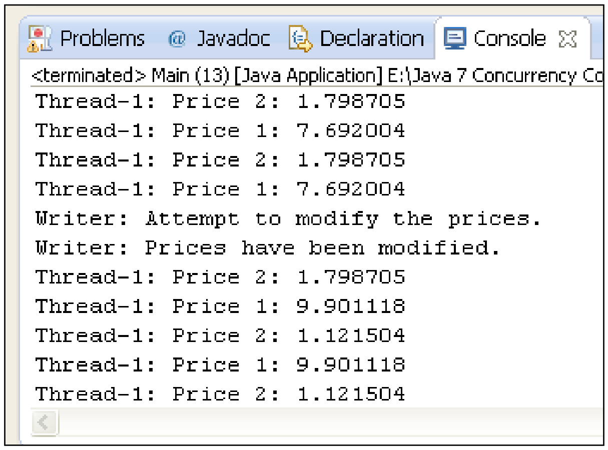
```
    Writer writer=new Writer(pricesInfo);
      Thread  threadWriter=new Thread(writer);
```

18. Start the threads.

```
    for (int i=0; i<5; i++){
      threadsReader[i].start();
    }
    threadWriter.start();
```

## How it works...

In the following screenshot, you can see a part of the output of one execution of this example:




As we mentioned previously, the `ReentrantReadWriteLock` class has two locks, one for read operations and one for write operations. The lock used in read operations is obtained with the `readLock()` method declared in the `ReadWriteLock` interface. This lock is an object that implements the `Lock` interface, so we can use the `lock()`, `unlock()`, and `tryLock()` methods. The lock used in write operations is obtained with the `writeLock()` method declared in the `ReadWriteLock` interface. This lock is an object that implements the `Lock` interface, so we can use the `lock()`, `unlock()`, and `tryLock()` methods. It is the responsibility of the programmer to ensure the correct use of these locks, using them with the same purposes for which they were designed.When you get the read lock of a `Lock` interface, you can't modify the value of the variable. Otherwise, you probably will have inconsistency data errors.

## See also

- The *Synchronizing a block of code with a Lock* recipe in *Chapter 2, Basic Thread Synchronization*
- The *Monitoring a Lock interface* recipe in *Chapter 8, Testing concurrent Applications*

# Modifying Lock fairness

The constructor of the `ReentrantLock` and `ReentrantReadWriteLock` classes admits a `boolean` parameter named `fair` that allows you to control the behavior of both classes. The `false` value is the default value and it's called the **non-fair mode**. In this mode, when there are some threads waiting for a lock (`ReentrantLock` or `ReentrantReadWriteLock`) and the lock has to select one of them to get the access to the critical section, it selects one without any criteria. The `true` value is called the **fair mode**. In this mode, when there are some threads waiting for a lock (`ReentrantLock` or `ReentrantReadWriteLock`) and the lock has to select one to get access to a critical section, it selects the thread that has been waiting for the most time. Take into account that the behavior explained previously is only used with the `lock()` and `unlock()` methods. As the `tryLock()` method doesn't put the thread to sleep if the `Lock` interface is used, the fair attribute doesn't affect its functionality.

In this recipe, we will modify the example implemented in the *Synchronizing a block of code with a Lock* recipe to use this attribute and see the difference between the fair and non-fair modes.

## Getting Ready...

We are going to modify the example implemented in the *Synchronizing a block of code with a Lock* recipe, so read that recipe to implement this example.

## How to do it...

Follow these steps to implement the example:

1. Implement the example explained in the *Synchronizing a block of code with a Lock* recipe.
2. In the `PrintQueue` class, modify the construction of the `Lock` object. The new instruction is given as follows:

```
private Lock queueLock=new ReentrantLock(true);
```

3. Modify the `printJob()` method. Separate the simulation of printing in two blocks of code, freeing the lock between them.

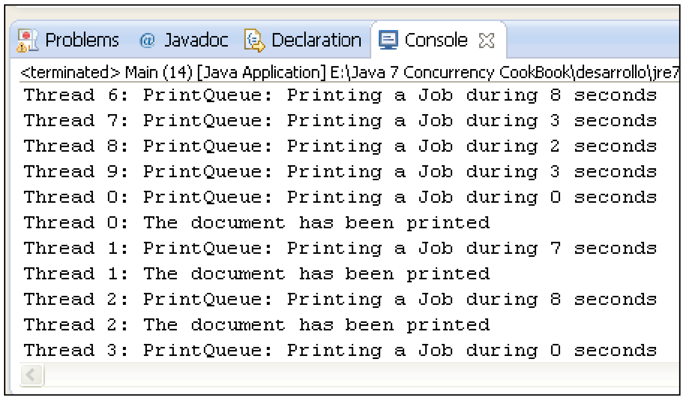
```
  public void printJob(Object document){
    queueLock.lock();
    try {
      Long duration=(long)(Math.random()*10000);
      System.out.println(Thread.currentThread().getName()+":
PrintQueue: Printing a Job during "+(duration/1000)+" seconds");
      Thread.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
       queueLock.unlock();
    }
    queueLock.lock();
    try {
      Long duration=(long)(Math.random()*10000);
      System.out.println(Thread.currentThread().getName()+":
PrintQueue: Printing a Job during "+(duration/1000)+" seconds");
      Thread.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
          queueLock.unlock();
       }
  }
```

4. Modify in the `Main` class the block of code that starts the threads. The new block of code is given as follows:

```
    for (int i=0; i<10; i++){
      thread[i].start();
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

## How it works...

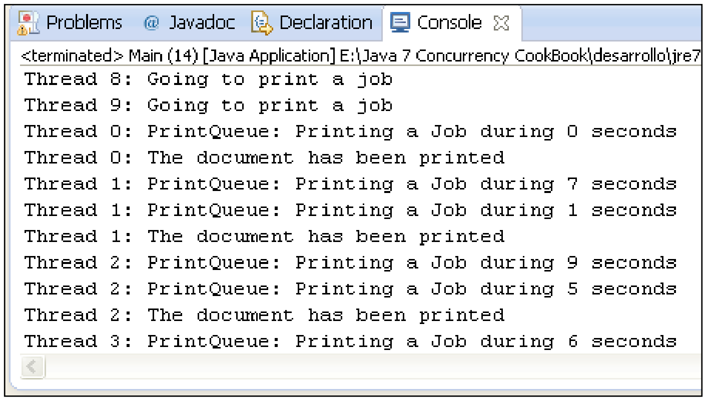
In the following screenshot you can see a part of the output of one execution of this example:




All threads are created with a difference of 0.1 seconds. The first thread that requests the control of the lock is **Thread 0**, then **Thread 1**, and so on. While **Thread 0** is running the first block of code protected by the lock, we have nine threads waiting to execute that block of code. When **Thread 0** releases the lock, immediately, it requests the lock again, so we have 10 threads trying to get the lock. As the fair mode is enabled, the `Lock` interface will choose **Thread 1**, so it's the thread that has been waiting for more time for the lock. Then, it chooses **Thread 2**, then, **Thread 3**, and so on. Until all the threads have passed the first block protected by the lock, none of them will execute the second block protected by the lock.

Once all the threads have executed the first block of code protected by the lock, it's the turn of **Thread 0** again. Then, it's the turn of **Thread 1**, and so on.

To see the difference with the non-fair mode, change the parameter passed to the lock constructor and put the `false` value. In the following screenshot, you can see the result of one execution of the modified example:




In this case, the threads are executed in the order that have been created but each thread executes the two protected blocks of code. However, this behavior is not guaranteed because, as explained earlier, the lock could choose any thread to give it access to the protected code. The JVM does not guarantee, in this case, the order of execution of the threads.

## There's more...

Read/write locks also have the fair parameter in their constructor. The behaviour of this parameter in this kind of lock is the same as we explained in the introduction of this recipe.

## See also

- The *Synchronizing a block of code with a Lock* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Synchronizing data access with read/write locks* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Implementing a custom Lock class* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Using multiple conditions in a Lock

A lock may be associated with one or more conditions. These conditions are declared in the `Condition` interface. The purpose of these conditions is to allow threads to have control of a lock and check whether a condition is `true` or not and, if it's `false`, be suspended until another thread wakes them up. The `Condition` interface provides the mechanisms to suspend a thread and to wake up a suspended thread.

A classic problem in concurrent programming is the **producer-consumer** problem. We have a data buffer, one or more **producers** of data that save it in the buffer, and one or more **consumers** of data that take it from the buffer as explained earlier in this chapter

In this recipe, you will learn how to implement the producer-consumer problem using locks and conditions.

## Getting Ready...

You should read the *Synchronizing a block of code with a Lock* recipe for a better understanding of this recipe.

## How to do it...

Follow these steps to implement the example:

1. First, let's implement a class that will simulate a text file. Create a class named `FileMock` with two attributes: a `String` array named `content` and `int` named `index`. They will store the content of the file and the line of the simulated file that will be retrieved.

```
public class FileMock {

  private String content[];
  private int index;
```

2. Implement the constructor of the class that initializes the content of the file with random characters.

```
  public FileMock(int size, int length){
    content=new String[size];
    for (int i=0; i<size; i++){
      StringBuilder buffer=new StringBuilder(length);
      for (int j=0; j<length; j++){
        int indice=(int)Math.random()*255;
        buffer.append((char)indice);
      }
      content[i]=buffer.toString();
    }
    index=0;
  }
```

3. Implement the method `hasMoreLines()` that returns `true` if the file has more lines to process or `false` if we have achieved the end of the simulated file.

```
  public boolean hasMoreLines(){
    return index<content.length;
  }
```

4. Implement the method `getLine()` that returns the line determined by the index attribute and increases its value.

```
public String getLine(){
  if (this.hasMoreLines()) {
    System.out.println("Mock: "+(content.length-index));
    return content[index++];
  }
  return null;
}
```

5. Now, implement a class named `Buffer` that will implement the buffer shared by producers and consumers.

```
public class Buffer {
```

6. This class has six attributes:
   - A `LinkedList<String>` attribute named `buffer` that will store the shared data
   - An `int` type named `maxSize` that stores the length of the buffer
   - A `ReentrantLock` object called `lock` that controls the access to the blocks of code that modify the buffer
   - Two `Condition` attributes named `lines` and `space`
   - A `boolean` type called `pendingLines` that will indicate if there are lines in the buffer

```
private LinkedList<String> buffer;

private int maxSize;

private ReentrantLock lock;

private Condition lines;
private Condition space;

private boolean pendingLines;
```

7. Implement the constructor of the class. It initializes all the attributes described previously.

```
public Buffer(int maxSize) {
  this.maxSize=maxSize;
  buffer=new LinkedList<>();
  lock=new ReentrantLock();
  lines=lock.newCondition();
  space=lock.newCondition();
  pendingLines=true;
}
```

8. Implement the `insert()` method. It receives `String` as a parameter and tries to store it in the buffer. First, it gets the control of the lock. When it has it, it then checks if there is empty space in the buffer. If the buffer is full, it calls the `await()` method in the `space` condition to wait for free space. The thread will be woken up when another thread calls the `signal()` or `signalAll()` method in the space `Condition`. When that happens, the thread stores the line in the buffer and calls the `signallAll()` method over the `lines` condition. As we'll see in a moment, this condition will wake up all the threads that were waiting for lines in the buffer.

```
  public void insert(String line) {
    lock.lock();
    try {
      while (buffer.size() == maxSize) {
        space.await();
      }
      buffer.offer(line);
      System.out.printf("%s: Inserted Line: %d\n", Thread.
currentThread().getName(),buffer.size());
      lines.signalAll();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      lock.unlock();
    }
  }
```

9. Implement the `get()` method. It returns the first string stored in the buffer. First, it gets the control of the lock. When it has it, it checks if there are lines in the buffer. If the buffer is empty, it calls the `await()` method in the `lines` condition to wait for lines in the buffer. This thread will be woken up when another thread calls the `signal()` or `signalAll()` method in the lines condition. When it happens, the method gets the first line in the buffer, calls the `signalAll()` method over the space condition and returns `String`.

```
  public String get() {
    String line=null;
    lock.lock();
    try {
      while ((buffer.size() == 0) &&(hasPendingLines())) {
        lines.await();
      }

      if (hasPendingLines()) {
        line = buffer.poll();
        System.out.printf("%s: Line Readed: %d\n",Thread.
currentThread().getName(),buffer.size());
```

```
        space.signalAll();
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      lock.unlock();
    }
    return line;
  }
```

10. Implement the `setPendingLines()` method that establishes the value of the attribute `pendingLines`. It will be called by the producer when it has no more lines to produce.

```
  public void setPendingLines(boolean pendingLines) {
    this.pendingLines=pendingLines;
  }
```

11. Implement the `hasPendingLines()` method. It returns `true` if there are more lines to be processed, or `false` otherwise.

```
  public boolean hasPendingLines() {
    return pendingLines || buffer.size()>0;
  }
```

12. It's now the turn of the producer. Implement a class named `Producer` and specify that it implements the `Runnable` interface.

```
public class Producer implements Runnable {
```

13. Declare two attributes: one object of the `FileMock` class and another object of the `Buffer` class.

```
  private FileMock mock;

  private Buffer buffer;
```

14. Implement the constructor of the class that initializes both attributes.

```
  public Producer (FileMock mock, Buffer buffer){
    this.mock=mock;
    this.buffer=buffer;
  }
```

15. Implement the `run()` method that reads all the lines created in the `FileMock` object and uses the `insert()` method to store them in the buffer. Once it finishes, use the `setPendingLines()` method to alert the buffer that it's not going to generate more lines.

```
 @Override
public void run() {
  buffer.setPendingLines(true);
  while (mock.hasMoreLines()){
    String line=mock.getLine();
    buffer.insert(line);
  }
  buffer.setPendingLines(false);
}
```

16. Next is the consumer's turn. Implement a class named `Consumer` and specify that it implements the `Runnable` interface.

```
public class Consumer implements Runnable {
```

17. Declare a `Buffer` object and implement the constructor of the class that initializes it.

```
private Buffer buffer;

public Consumer (Buffer buffer) {
  this.buffer=buffer;
}
```

18. Implement the `run()` method. While the buffer has pending lines, it tries to get one and process it.

```
 @Override
public void run() {
  while (buffer.hasPendingLines()) {
    String line=buffer.get();
    processLine(line);
  }
}
```

19. Implement the auxiliary method `processLine()`. It only sleeps for 10 milliseconds to simulate some kind of processing with the line.

```
private void processLine(String line) {
  try {
    Random random=new Random();
    Thread.sleep(random.nextInt(100));
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

20. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

21. Create a `FileMock` object.

```
    FileMock mock=new FileMock(100, 10);
```

22. Create a `Buffer` object.

```
    Buffer buffer=new Buffer(20);
```

23. Create a `Producer` object and `Thread` to run it.

```
    Producer producer=new Producer(mock, buffer);
    Thread threadProducer=new Thread(producer,"Producer");
```

24. Create three `Consumer` objects and three threads to run it.

```
    Consumer consumers[]=new Consumer[3];
    Thread threadConsumers[]=new Thread[3];

    for (int i=0; i<3; i++){
      consumers[i]=new Consumer(buffer);
      threadConsumers[i]=new Thread(consumers[i],"Consumer "+i);
    }
```

25. Start the producer and the three consumers.

```
    threadProducer.start();
    for (int i=0; i<3; i++){
      threadConsumers[i].start();
    }
```

## How it works...

All the `Condition` objects are associated with a lock and are created using the `newCondition()` method declared in the `Lock` interface. Before we can do any operation with a condition, you have to have the control of the lock associated with the condition, so the operations with conditions must be in a block of code that begins with a call to a `lock()` method of a `Lock` object and ends with an `unlock()` method of the same `Lock` object.

When a thread calls the `await()` method of a condition, it automatically frees the control of the lock, so that another thread can get it and begin the execution of the same, or another critical section protected by that lock.

> When a thread calls the `signal()` or `signallAll()` methods of a condition, one or all of the threads that were waiting for that condition are woken up, but this doesn't guarantee that the condition that made them sleep is now `true`, so you must put the `await()` calls inside a `while` loop. You can't leave that loop until the condition is `true`. While the condition is `false`, you must call `await()` again.

You must be careful with the use of `await()` and `signal()`. If you call the `await()` method in a condition and never call the `signal()` method in this condition, the thread will be sleeping forever.

A thread can be interrupted while it is sleeping, after a call to the `await()` method, so you have to process the `InterruptedException` exception.

## There's more...

The `Condition` interface has other versions of the `await()` method, which are as follows:

- `await(long time, TimeUnit unit)`: The thread will be sleeping until:
    - It's interrupted
    - Another thread calls the `singal()` or `signalAll()` methods in the condition
    - The specified time passes
    - The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`
- `awaitUninterruptibly()`: The thread will be sleeping until another thread calls the `signal()` or `signalAll()` methods, which can't be interrupted
- `awaitUntil(Date date)`: The thread will be sleeping until:
    - It's interrupted
    - Another thread calls the `singal()` or `signalAll()` methods in the condition
    - The specified date arrives

You can use conditions with the `ReadLock` and `WriteLock` locks of a read/write lock.

## See also

- The *Synchronizing a block of code with a Lock* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Synchronizing data access with read/write locks* recipe in *Chapter 2*, *Basic Thread Synchronization*

# 3

# Thread Synchronization Utilities

In this chapter, we will cover:

- Controlling concurrent access to a resource
- Controlling concurrent access to multiple copies of a resource
- Waiting for multiple concurrent events
- Synchronizing tasks in a common point
- Running concurrent phased tasks
- Controlling phase change in concurrent phased tasks
- Changing data between concurrent tasks

## Introduction

In *Chapter 2*, *Basic thread synchronization*, we learned the concepts of synchronization and critical section. Basically, we talk about synchronization when more than one concurrent task shares a resource, for example, an object or an attribute of an object. The blocks of code that access this shared resource are called critical sections.

If you don't use the appropriate mechanisms, you can have the wrong results, data inconsistency, or error conditions, so we have to adopt one of the synchronization mechanisms provided by the Java language to avoid all these problems.

*Chapter 2*, *Basic thread synchronization*, taught us about the following basic synchronization mechanisms:

- The `synchronized` keyword
- The `Lock` interface and its implementation classes: `ReentrantLock`, `ReentrantReadWriteLock.ReadLock`, and `ReentrantReadWriteLock.WriteLock`

In this chapter, we will learn how to use high-level mechanisms to get the synchronization of multiple threads. These high-level mechanisms are as follows:

- **Semaphores**: A semaphore is a counter that controls the access to one or more shared resources. This mechanism is one of the basic tools of concurrent programming and is provided by most of the programming languages.
- **CountDownLatch**: The `CountDownLatch` class is a mechanism provided by the Java language that allows a thread to wait for the finalization of multiple operations.
- **CyclicBarrier**: The `CyclicBarrier` class is another mechanism provided by the Java language that allows the synchronization of multiple threads in a common point.
- **Phaser**: The `Phaser` class is another mechanism provided by the Java language that controls the execution of concurrent tasks divided in phases. All the threads must finish one phase before they can continue with the next one. This is a new feature of the Java 7 API.
- **Exchanger**: The `Exchanger` class is another mechanism provided by the Java language that provides a point of data interchange between two threads.

Semaphores are a generic synchronization mechanism that you can use to protect any critical section in any problem. The other mechanisms are thought to be used in applications with specific features as it was described previously. Be sure to select the appropriate mechanism according to the characteristics of your application.

This chapter presents seven recipes that show you how to use the mechanisms described.

# Controlling concurrent access to a resource

In this recipe, you will learn how to use the semaphore mechanism provided by the Java language. A semaphore is a counter that protects the access to one or more shared resources.

> The concept of a semaphore was introduced by Edsger Dijkstra in 1965 and was used for the first time in the THEOS operating system.

When a thread wants to access one of these shared resources, first, it must acquire the semaphore. If the internal counter of the semaphore is greater than `0`, the semaphore decrements the counter and allows access to the shared resource. A counter bigger than `0` means there are free resources that can be used, so the thread can access and use one of them.

Otherwise, if the counter of the semaphore is `0`, the semaphore puts the thread to sleep until the counter is greater than `0`. A value of `0` in the counter means all the shared resources are used by other threads, so the thread that wants to use one of them must wait until one is free.

When the thread has finished the use of the shared resource, it must release the semaphore so that the other thread can access the shared resource. That operation increases the internal counter of the semaphore.

In this recipe, you will learn how to use the `Semaphore` class to implement special kinds of semaphores called **binary semaphores**. These kinds of semaphores protect the access to a unique shared resource, so the internal counter of the semaphore can only take the values `1` or `0`. To show how to use it, you are going to implement a print queue that can be used by concurrent tasks to print their jobs. This print queue will be protected by a binary semaphore, so only one thread can print at a time.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `PrintQueue` that will implement the print queue.

   ```
   public class PrintQueue {
   ```

2. Declare a `Semaphore` object. Call it `semaphore`.

   ```
     private final Semaphore semaphore;
   ```

3. Implement the constructor of the class. It initializes the `semaphore` object that will protect the access from the print queue.

   ```
     public PrintQueue(){
       semaphore=new Semaphore(1);
     }
   ```

4. Implement the `printJob()` method that will simulate the printing of a document. It receives `Object` called `document` as a parameter.

```
public void printJob (Object document){
```

5. Inside the method, first of all, you must acquire the semaphore calling the `acquire()` method. This method can throw an `InterruptedException` exception, so you must include some code to process it.

```
    try {
      semaphore.acquire();
```

6. Then, implement the lines that simulate the printing of a document waiting for a random period of time.

```
  long duration=(long)(Math.random()*10);
      System.out.printf("%s: PrintQueue: Printing a Job during %d 
seconds\n",Thread.currentThread().getName(),duration);
      Thread.sleep(duration);
```

7. Finally, free the semaphore by calling the `release()` method of the semaphore.

```
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      semaphore.release();
    }
```

8. Create a class called `Job` and specify that it implements the `Runnable` interface. This class implements a job that sends a document to the printer.

```
public class Job implements Runnable {
```

9. Declare a `PrintQueue` object. Call it `printQueue`.

```
  private PrintQueue printQueue;
```

10. Implement the constructor of the class. It initializes the `PrintQueue` object declared in the class.

```
  public Job(PrintQueue printQueue){
    this.printQueue=printQueue;
  }
```

11. Implement the `run()` method.

```
  @Override
   public void run() {
```

12. First, the method writes a message to the console that shows that the job has started its execution.

```
    System.out.printf("%s: Going to print a job\n",Thread.
currentThread().getName());
```

13. Then, it calls the `printJob()` method of the `PrintQueue` object.

```
    printQueue.printJob(new Object());
```

14. Finally, the method writes a message to the console that shows that it has finished its execution.

```
    System.out.printf("%s: The document has been printed\
n",Thread.currentThread().getName());
  }
```

15. Implement the main class of the example by creating a class named `Main` and implement the `main()` method.

```
public class Main {

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

16. Create a `PrintQueue` object named `printQueue`.

```
    PrintQueue printQueue=new PrintQueue();
```

17. Create 10 threads. Each one of those threads will execute a `Job` object that will send a document to the print queue.

```
    Thread thread[]=new Thread[10];
    for (int i=0; i<10; i++){
      thread[i]=new Thread(new Job(printQueue),"Thread"+i);
    }
```

18. Finally, start the 10 threads.

```
    for (int i=0; i<10; i++){
      thread[i].start();
    }
```

## How it works...

The key to this example is in the `printJob()` method of the `PrintQueue` class. This method shows the three steps you must follow when you use a semaphore to implement a critical section, and protect the access to a shared resource:

1. First, you acquire the semaphore, with the `acquire()` method.
2. Then, you do the necessary operations with the shared resource.
3. Finally, release the semaphore with the `release()` method.

Another important point in this example is the constructor of the `PrintQueue` class and the initialization of the `Semaphore` object. You pass the value `1` as the parameter of this constructor, so you are creating a binary semaphore. The initial value of the internal counter is `1`, so you will protect the access to one shared resource, in this case, the print queue.

When you start the 10 threads, the first one acquires the semaphore and gets the access to the critical section. The rest are blocked by the semaphore until the thread that has acquired it, releases it. When this occurs, the semaphore selects one of the waiting threads and gives it the access to the critical section. All the jobs print their documents, but one by one.

## There's more...

The `Semaphore` class has two additional versions of the `acquire()` method:

- `acquireUninterruptibly()`: The `acquire()` method; when the internal counter of the semaphore is `0`, blocks the thread until the semaphore is released. During this blocked time, the thread may be interrupted and then this method throws an `InterruptedException` exception. This version of the acquire operation ignores the interruption of the thread and doesn't throw any exceptions.
- `tryAcquire()`: This method tries to acquire the semaphore. If it can, the method returns the `true` value. But if it can't, the method returns the `false` value instead of being blocked and waits for the release of the semaphore. It's your responsibility to take the correct action based on the `return` value.

### Fairness in semaphores

The concept of fairness is used by the Java language in all classes that can have various threads blocked waiting for the release of a synchronization resource (for example, a semaphore). The default mode is called the **non-fair mode**. In this mode, when the synchronization resource is released, one of the waiting threads is selected to get this resource, but it's selected without any criteria. The **fair mode** changes this behavior and forces to select the thread that has been waiting for more time.

As occurs with other classes, the `Semaphore` class admits a second parameter in its constructor. This parameter must take a `Boolean` value. If you give it the `false` value, you are creating a semaphore that will work in non-fair mode. You will get the same behavior if you don't use this parameter. If you give it the `true` value, you are creating a semaphore that will work in fair mode.

## See also

- The *Monitoring a Lock interface* recipe in *Chapter 8*, *Testing Concurrent Applications*
- The *Modifying Lock fairness* recipe in *Chapter 2*, *Basic Thread Synchronization*

# Controlling concurrent access to multiple copies of a resource

In the *Controlling concurrent access to a resource* recipe, you learned the basis of semaphores.

In that recipe, you implemented an example using binary semaphores. These kinds of semaphores are used to protect the access to one shared resource, or to a critical section that can only be executed by one thread at a time. But semaphores can also be used when you need to protect various copies of a resource, or when you have a critical section that can be executed by more than one thread at the same time.

In this recipe, you will learn how to use a semaphore to protect more than one copy of a resource. You are going to implement an example, which has one print queue that can print documents in three different printers.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

Implement the example described in the *Controlling concurrent access to a resource* recipe in this chapter.

## How to do it...

Follow these steps to implement the example:

1. As we mentioned earlier, you are going to modify the print queue example implemented with semaphores. Open the `PrintQueue` class and declare a `boolean` array called `freePrinters`. This array stores printers that are free to print a job and printers that are printing a document.

   ```
   private boolean freePrinters[];
   ```

2. Also, declare a `Lock` object named `lockPrinters`. You will use this object to protect the access to the `freePrinters` array.

   ```
   private Lock lockPrinters;
   ```

3. Modify the constructor of the class to initialize the new declared objects. The `freePrinters` array has three elements, all initialized to the `true` value. The semaphore has `3` as its initial value.

   ```
   public PrintQueue(){
     semaphore=new Semaphore(3);
     freePrinters=new boolean[3];
   ```

```
    for (int i=0; i<3; i++){
      freePrinters[i]=true;
    }
    lockPrinters=new ReentrantLock();
  }
```

4. Modify also the `printJob()` method. It receives an `Object` called `document` as the unique parameter.

```
  public void printJob (Object document){
```

5. First of all, the method calls the `acquire()` method to acquire the access to the semaphore. As this method can throw an `InterruptedException` exception, you must include the code to process it.

```
    try {
      semaphore.acquire();
```

6. Then you get the number of the printer assigned to print this job using the private method `getPrinter()`.

```
      int assignedPrinter=getPrinter();
```

7. Then, implement the lines that simulate the printing of a document waiting for a random period of time.

```
      long duration=(long)(Math.random()*10);
      System.out.printf("%s: PrintQueue: Printing a Job in Printer
      %d during %d seconds\n",Thread.currentThread().getName(),
      assignedPrinter,duration);
      TimeUnit.SECONDS.sleep(duration);
```

8. Finally, release the semaphore calling the `release()` method and mark the printer used as free, assigning `true` to the corresponding index in the `freePrinters` array.

```
      freePrinters[assignedPrinter]=true;
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      semaphore.release();
    }
```

9. Implement the `getPrinter()` method. It's a private method that returns an `int` value and it has no parameters.

```
  private int getPrinter() {
```

10. First of all, declare an `int` variable to store the index of the printer.

```
    int ret=-1;
```

11. Then, get the access to the `lockPrinters` object.

```
try {
  lockPrinters.lock();
```

12. Then, find the first `true` value in the `freePrinters` array and save its index in a variable. Modify this value to `false`, because this printer will be busy.

```
for (int i=0; i<freePrinters.length; i++) {
  if (freePrinters[i]){
    ret=i;
    freePrinters[i]=false;
    break;
  }
}
```

13. Finally, free the `lockPrinters` object and return the index of the `true` value.

```
} catch (Exception e) {
  e.printStackTrace();
} finally {
  lockPrinters.unlock();
}
return ret;
```

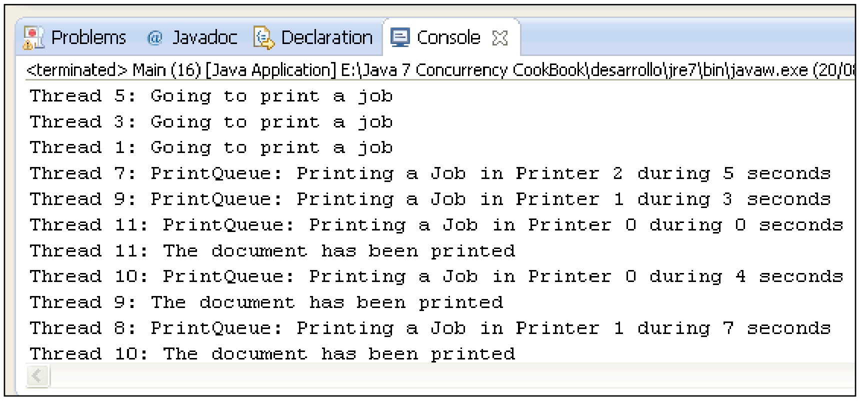
14. The `Job` and `Core` classes have no modifications.

## How it works...

The key of this example is in the `PrintQueue` class. The `Semaphore` object is created using `3` as the parameter of the constructor. The first three threads that call the `acquire()` method will get the access to the critical section of this example, while the rest will be blocked. When a thread finishes the critical section and releases the semaphore, another thread will acquire it.

In this critical section, the thread gets the index of the printer assigned to print this job. This part of the example is used to give more realism to the example, but it doesn't use any code related with semaphores.

The following screenshot shows the output of an execution of this example:




Each document is printed in one of the printers. The first one is free.

## There's more...

The `acquire()`, `acquireUninterruptibly()`, `tryAcquire()`, and `release()` methods have an additional version which has an `int` parameter. This parameter represents the number of permits that the thread that uses them wants to acquire or release, so as to say, the number of units that this thread wants to delete or to add to the internal counter of the semaphore. In the case of the `acquire()`, `acquireUninterruptibly()`, and `tryAcquire()` methods, if the value of this counter is less than this value, the thread will be blocked until the counter gets this value or a greater one.

## See also

- The *Controlling concurrent access to a resource* recipe in *Chapter 3*, *Thread Synchronization Utilities*
- The *Monitoring a Lock interface* recipe in *Chapter 8*, *Testing Concurrent Applications*
- The *Modifying lock fairness* recipe in *Chapter 2*, *Basic Thread Synchronization*

# Waiting for multiple concurrent events

The Java concurrency API provides a class that allows one or more threads to wait until a set of operations are made. It's the `CountDownLatch` class. This class is initialized with an integer number, which is the number of operations the threads are going to wait for. When a thread wants to wait for the execution of these operations, it uses the `await()` method. This method puts the thread to sleep until the operations are completed. When one of these operations finishes, it uses the `countDown()` method to decrement the internal counter of the `CountDownLatch` class. When the counter arrives to `0`, the class wakes up all the threads that were sleeping in the `await()` method.

In this recipe, you will learn how to use the `CountDownLatch` class implementing a video-conference system. The video-conference system will wait for the arrival of all the participants before it begins.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Videoconference` and specify that it implements the `Runnable` interface. This class will implement the video-conference system.

```
public class Videoconference implements Runnable{
```

2. Declare a `CountDownLatch` object named `controller`.

```
  private final CountDownLatch controller;
```

3. Implement the constructor of the class that initializes the `CountDownLatch` attribute. The `Videoconference` class will wait for the arrival of the number of participants received as a parameter.

```
  public Videoconference(int number) {
    controller=new CountDownLatch(number);
  }
```

4. Implement the `arrive()` method. This method will be called each time a participant arrives to the video conference. It receives a `String` type named `name` as the parameter.

```
  public void arrive(String name){
```

5. First, it writes a message with the parameter it has received.

```
    System.out.printf("%s has arrived.",name);
```

6. Then, it calls the `countDown()` method of the `CountDownLatch` object.

```
    controller.countDown();
```

7. Finally, it writes another message with the number of participants, whose arrival is pending using the `getCount()` method of the `CountDownLatch` object.

```
    System.out.printf("VideoConference: Waiting for %d
participants.\n",controller.getCount());
```

8. Implement the main method of the video-conference system. It's the `run()` method that every `Runnable` object must have.

```
  @Override
  public void run() {
```

9. First, use the `getCount()` method to write a message with the number of participants in the video conference.

```
    System.out.printf("VideoConference: Initialization: %d
participants.\n",controller.getCount());
```

10. Then, use the `await()` method to wait for all the participants. As this method can throw an `InterruptedException` exception, you must include the code to process it.

```
    try {
      controller.await();
```

11. Finally, write a message to indicate that all the participants have arrived.

```
      System.out.printf("VideoConference: All the participants
have come\n");
      System.out.printf("VideoConference: Let's start...\n");
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

12. Create the `Participant` class and specify that it implements the `Runnable` interface. This class represents each participant in the video conference.

```
public class Participant implements Runnable {
```

13. Declare a private `Videoconference` attribute named `conference`.

```
  private Videoconference conference;
```

14. Declare a private `String` attribute named `name`.

```
  private String name;
```

15. Implement the constructor of the class that initializes both attributes.

```
  public Participant(Videoconference conference, String name) {
    this.conference=conference;
    this.name=name;
  }
```

16. Implement the `run()` method of the participants.

```
 @Override
public void run() {
```

17. First, put the thread to sleep for a random period of time.

```
long duration=(long)(Math.random()*10);
try {
  TimeUnit.SECONDS.sleep(duration);
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

18. Then, use the `arrive()` method of the `Videoconference` object to indicate the arrival of this participant.

```
conference.arrive(name);
```

19. Finally, implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

20. Create a `Videoconference` object named `conference` that waits for 10 participants.

```
Videoconference conference=new Videoconference(10);
```

21. Create `Thread` to run this `Videoconference` object and start it.

```
Thread threadConference=new Thread(conference);
threadConference.start();
```

22. Create 10 `Participant` objects, a `Thread` object to run each of them, and start all the threads.

```
for (int i=0; i<10; i++){
  Participant p=new Participant(conference, "Participant "+i);
  Thread t=new Thread(p);
  t.start();
}
```

## How it works...

The `CountDownLatch` class has three basic elements:

- The initialization value that determines how many events the `CountDownLatch` class waits for
- The `await()` method, called by the threads that wait for the finalization of all the events
- The `countDown()` method, called by the events when they finish their execution

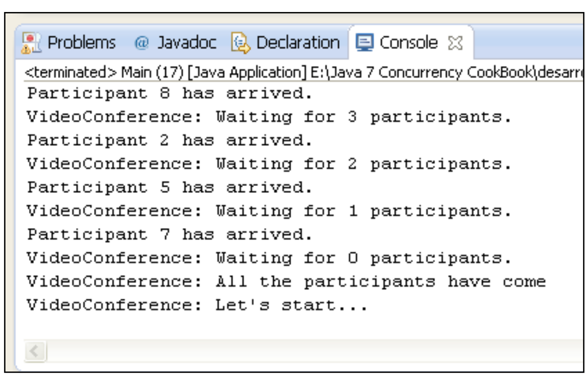
When you create a `CountDownLatch` object, the object uses the constructor's parameter to initialize an internal counter. Every time a thread calls the `countDown()` method, the `CountDownLatch` object decrements the internal counter in one unit. When the internal counter arrives to `0`, the `CountDownLatch` object wakes up all the threads that were waiting in the `await()` method.

There's no way to re-initialize the internal counter of the `CountDownLatch` object or to modify its value. Once the counter is initialized, the only method you can use to modify its value is the `countDown()` method explained earlier. When the counter arrives to `0`, all the calls to the `await()` method return immediately and all subsequent calls to the `countDown()` method have no effect.

There are some differences with respect to other synchronization methods, which are as follows:

- The `CountDownLatch` mechanism is not used to protect a shared resource or a critical section. It is used to synchronize one or more threads with the execution of various tasks.
- It only admits one use. As we explained earlier, once the counter of `CountDownLatch` arrives at `0`, all the calls to its methods have no effect. You have to create a new object if you want to do the same synchronization again.

The following screenshot shows the output of an execution of the example:



You can see how the last participants arrive and, once the internal counter arrives to `0`, the `CountDownLatch` object wakes up the `Videoconference` object that writes the messages indicating that the video conference should start.

## There's more...

The `CountDownLatch` class has another version of the `await()` method, which is given as follows:

- `await(long time, TimeUnit unit)`: The thread will be sleeping until it's interrupted; the internal counter of `CountDownLatch` arrives to `0` or specified time passes. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

# Synchronizing tasks in a common point

The Java concurrency API provides a synchronizing utility that allows the synchronization of two or more threads in a determined point. It's the `CyclicBarrier` class. This class is similar to the `CountDownLatch` class explained in the *Waiting for multiple concurrent events* recipe in this chapter, but presents some differences that make them a more powerful class.

The `CyclicBarrier` class is initialized with an integer number, which is the number of threads that will be synchronized in a determined point. When one of those threads arrives to the determined point, it calls the `await()` method to wait for the other threads. When the thread calls that method, the `CyclicBarrier` class blocks the thread that is sleeping until the other threads arrive. When the last thread calls the `await()` method of the `CyclicBarrier` class, it wakes up all the threads that were waiting and continues with its job.

One interesting advantage of the `CyclicBarrier` class is that you can pass an additional `Runnable` object as an initialization parameter, and the `CyclicBarrier` class executes this object as a thread when all the threads have arrived to the common point. This characteristic makes this class adequate for the parallelization of tasks using the divide and conquer programming technique.

In this recipe, you will learn how to use the `CyclicBarrier` class to synchronize a set of threads in a determined point. You will also use a `Runnable` object that will execute after all the threads have arrived to that point. In the example, you will look for a number in a matrix of numbers. The matrix will be divided in subsets (using the divide and conquer technique), so each thread will look for the number in one subset. Once all the threads have finished their job, a final task will unify the results of them.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. We're going to start the example by implementing two auxiliary classes. First, create a class named `MatrixMock`. This class will generate a random matrix of numbers between one and 10 where the threads are going to look for a number.

   ```
   public class MatrixMock {
   ```

2. Declare a `private int` matrix named `data`.

   ```
     private int data[][];
   ```

3. Implement the constructor of the class. This constructor will receive the number of rows of the matrix, the length of each row, and the number we are going to look for as parameters. All the three parameters are of type `int`.

   ```
     public MatrixMock(int size, int length, int number){
   ```

4. Initialize the variables and objects used in the constructor.

   ```
       int counter=0;
       data=new int[size][length];
       Random random=new Random();
   ```

5. Fill the matrix with random numbers. Each time you generate a number, compare it with the number you are going to look for. If they are equal, increment the counter.

   ```
       for (int i=0; i<size; i++) {
         for (int j=0; j<length; j++){
           data[i][j]=random.nextInt(10);
           if (data[i][j]==number){
             counter++;
           }
         }
       }
   ```

6. Finally, print a message in the console, which shows the number of occurrences of the number you are going to look for in the generated matrix. This message will be used to check that the threads get the correct result.

   ```
       System.out.printf("Mock: There are %d ocurrences of number in
   generated data.\n",counter,number);
   ```

7. Implement the `getRow()` method. This method receives an `int` parameter with the number of a row in the matrix and returns the row if it exists, and returns `null` if it doesn't exist.

```
public int[] getRow(int row){
  if ((row>=0)&&(row<data.length)){
    return data[row];
  }
  return null;
}
```

8. Now, implement a class named `Results`. This class will store, in an array, the number of occurrences of the searched number in each row of the matrix.

```
public class Results {
```

9. Declare a private `int` array named `data`.

```
private int data[];
```

10. Implement the constructor of the class. This constructor receives an integer parameter with the number of elements of the array.

```
public Results(int size){
  data=new int[size];
}
```

11. Implement the `setData()` method. This method receives a position in the array and a value as parameters, and establishes the value of that position in the array.

```
public void  setData(int position, int value){
  data[position]=value;
}
```

12. Implement the `getData()` method. This method returns the array with the array of the results.

```
public int[] getData(){
  return data;
}
```

13. Now that you have the auxiliary classes, it's time to implement the threads. First, implement the `Searcher` class. This class will look for a number in determined rows of the matrix of random numbers. Create a class named `Searcher` and specify that it implements the `Runnable` interface.

```
public class Searcher implements Runnable {
```

14. Declare two private `int` attributes named `firstRow` and `lastRow`. These two attributes will determine the subset of rows where this object will look for.

```
private int firstRow;

private int lastRow;
```

15. Declare a private `MatrixMock` attribute named `mock`.

```
private MatrixMock mock;
```

16. Declare a private `Results` attribute named `results`.

```
private Results results;
```

17. Declare a private `int` attribute named `number` that will store the number we are going to look for.

```
private int number;
```

18. Declare a `CyclicBarrier` object named `barrier`.

```
private final CyclicBarrier barrier;
```

19. Implement the constructor of the class that initializes all the attributes declared before.

```
  public Searcher(int firstRow, int lastRow, NumberMock mock,
Results results, int number, CyclicBarrier barrier){
    this.firstRow=firstRow;
    this.lastRow=lastRow;
    this.mock=mock;
    this.results=results;
    this.number=number;
    this.barrier=barrier;
  }
```

20. Implement the `run()` method that will search for the number. It uses an internal variable called `counter` that will store the number of occurrences of the number in each row.

```
   @Override
  public void run() {
    int counter;
```

21. Print a message in the console with the rows assigned to this object.

```
    System.out.printf("%s: Processing lines from %d to %d.\
n",Thread.currentThread().getName(),firstRow,lastRow);
```

22. Process all the rows assigned to this thread. For each row, count the number of occurrences of the number you are searching for and store this number in the corresponding position of the `Results` object.

```
    for (int i=firstRow; i<lastRow; i++){
      int row[]=mock.getRow(i);
      counter=0;
      for (int j=0; j<row.length; j++){
        if (row[j]==number){
          counter++;
        }
      }
      results.setData(i, counter);
    }
```

23. Print a message in the console to indicate that this object has finished searching.

```
    System.out.printf("%s: Lines processed.\n",Thread.
currentThread().getName());
```

24. Call the `await()` method of the `CyclicBarrier` object and add the necessary code to process the `InterruptedException` and `BrokenBarrierException` exceptions that this method can throw.

```
    try {
      barrier.await();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (BrokenBarrierException e) {
      e.printStackTrace();
    }
```

25. Now, implement the class that calculates the total number of occurrences of the number in the matrix. It uses the `Results` object that stores the number of appearances of the number in each row of the matrix to make the calculation. Create a class named `Grouper` and specify that it implements the `Runnable` interface.

```
public class Grouper implements Runnable {
```

26. Declare a private `Results` attribute named `results`.

```
  private Results results;
```

27. Implement the constructor of the class that initializes the `Results` attribute.

```
  public Grouper(Results results){
    this.results=results;
  }
```

28. Implement the `run()` method that will calculate the total number of occurrences of the number in the array of results.

```
 @Override
public void run() {
```

29. Declare an `int` variable and write a message to the console to indicate the start of the process.

```
int finalResult=0;
System.out.printf("Grouper: Processing results...\n");
```

30. Get the number of occurrences of the number in each row using the `getData()` method of the `results` object. Then, process all the elements of the array and add their value to the `finalResult` variable.

```
int data[]=results.getData();
for (int number:data){
  finalResult+=number;
}
```

31. Print the result in the console.

```
System.out.printf("Grouper: Total result: %d.\n",finalResult);
```

32. Finally, implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

33. Declare and initialize five constants to store the parameters of the application.

```
final int ROWS=10000;
final int NUMBERS=1000;
final int SEARCH=5;
final int PARTICIPANTS=5;
final int LINES_PARTICIPANT=2000;
```

34. Create a `MatrixMock` object named `mock`. It will have 10,000 rows of 1000 elements. Now, you are going to search for the number five.

```
MatrixMock mock=new MatrixMock(ROWS, NUMBERS,SEARCH);
```

35. Create a `Results` object named `results`. It will have 10,000 elements.

```
Results results=new Results(ROWS);
```

36. Create a `Grouper` object named `grouper`.

```
Grouper grouper=new Grouper(results);
```

37. Create a `CyclicBarrier` object called `barrier`. This object will wait for five threads. When this thread finishes, it will execute the `Grouper` object created previously.

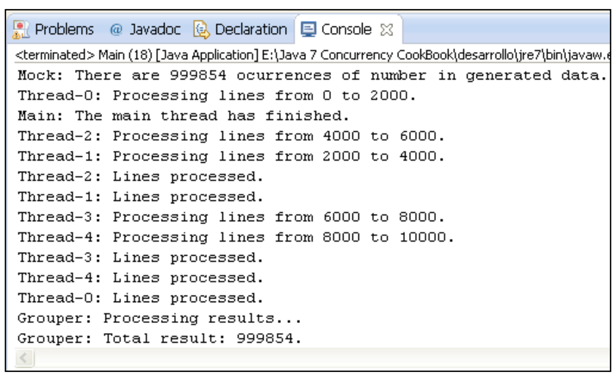
```
CyclicBarrier barrier=new CyclicBarrier(PARTICIPANTS,grouper);
```

38. Create five `Searcher` objects, five threads to execute them, and start the five threads.

```
    Searcher searchers[]=new Searcher[PARTICIPANTS];
    for (int i=0; i<PARTICIPANTS; i++){
      searchers[i]=new Searcher(i*LINES_PARTICIPANT, (i*LINES_
PARTICIPANT)+LINES_PARTICIPANT, mock, results, 5,barrier);
      Thread thread=new Thread(searchers[i]);
      thread.start();
    }
    System.out.printf("Main: The main thread has finished.\n");
```

## How it works...

The following screenshot shows the results of an execution of this example:




The problem resolved in the example is simple. We have a big matrix of random integer numbers and you want to know the total number of occurrences of a number in this matrix. To get a better performance, we use the divide and conquer technique. We divide the matrix in five subsets and use a thread to look for the number in each subset. These threads are objects of the `Searcher` class.

We use a `CyclicBarrier` object to synchronize the completion of the five threads and to execute the `Grouper` task to process the partial results, and calculate the final one.

As we mentioned earlier, the `CyclicBarrier` class has an internal counter to control how many threads have to arrive to the synchronization point. Each time a thread arrives to the synchronization point, it calls the `await()` method to notify the `CyclicBarrier` object that has arrived to its synchronization point. `CyclicBarrier` puts the thread to sleep until all the threads arrive to their synchronization point.

When all the threads have arrived to their synchronization point, the `CyclicBarrier` object wakes up all the threads that were waiting in the `await()` method and, optionally, creates a new thread that executes a `Runnable` object passed as the parameter in the construction of `CyclicBarrier` (in our case, a `Grouper` object) to do additional tasks.

## There's more...

The `CyclicBarrier` class has another version of the `await()` method:

- `await(long time, TimeUnit unit)`: The thread will be sleeping until it's interrupted; the internal counter of `CyclicBarrier` arrives to `0` or specified time passes. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

This class also provides the `getNumberWaiting()` method that returns the number of threads that are blocked in the `await()` method, and the `getParties()` method that returns the number of tasks that are going to be synchronized with `CyclicBarrier`.

### Resetting a CyclicBarrier object

The `CyclicBarrier` class has some points in common with the `CountDownLatch` class, but they also have some differences. One of the most important differences is that a `CyclicBarrier` object can be reset to its initial state, assigning to its internal counter the value with which it was initialized.

This reset operation can be done using the `reset()` method of the `CyclicBarrier` class. When this occurs, all the threads that were waiting in the `await()` method receive a `BrokenBarrierException` exception. This exception was processed in the example presented in this recipe by printing the stack trace, but in a more complex application, it could perform some other operation, such as restarting their execution or recovering their operation at the point it was interrupted.

### Broken CyclicBarrier objects

A `CyclicBarrier` object can be in a special state denoted by **broken**. When there are various threads waiting in the `await()` method and one of them is interrupted, this thread receives an `InterruptedException` exception, but the other threads that were waiting receive a `BrokenBarrierException` exception and `CyclicBarrier` is placed in the broken state.

The `CyclicBarrier` class provides the `isBroken()` method, then returns `true` if the object is in the broken state; otherwise it returns `false`.

## See also

- The *Waiting for multiple concurrent events* recipe in *Chapter 3*, *Thread Synchronization Utilities*

# Running concurrent phased tasks

One of the most complex and powerful functionalities offered by the Java concurrency API is the ability to execute concurrent-phased tasks using the `Phaser` class. This mechanism is useful when we have some concurrent tasks divided into steps. The `Phaser` class provides us with the mechanism to synchronize the threads at the end of each step, so no thread starts its second step until all the threads have finished the first one.

As with other synchronization utilities, we have to initialize the `Phaser` class with the number of tasks that participate in the synchronization operation,  but we can dynamically modify this number by increasing or decreasing it.

In this recipe, you will learn how to use the `Phaser` class to synchronize three concurrent tasks. The three tasks look for files with the extension `.log` modified in the last 24 hours in three different folders and their subfolders. This task is divided into three steps:

1. Get a list of the files with the extension `.log` in the assigned folder and its subfolders.
2. Filter the list created in the first step by deleting the files modified more than 24 hours ago.
3. Print the results in the console.

At the end of the steps 1 and 2 we check if the list has any elements or not. If it hasn't any element, the thread ends its execution and is eliminated from the the `phaser` class.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE like NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `FileSearch` and specify that it implements the `Runnable` interface. This class implements the operation of searching for files with a determined extension modified in the last 24 hours in a folder and its subfolders.

```
public class FileSearch implements Runnable {
```

2. Declare a private `String` attribute to store the folder in which the search operation will begin.

```
  private String initPath;
```

3. Declare another private `String` attribute to store the extension of the files we are going to look for.

```
  private String end;
```

4. Declare a private `List` attribute to store the full path of the files we will find with the desired characteristics.

```
  private List<String> results;
```

5. Finally, declare a private `Phaser` attribute to control the synchronization of the different phases of the task.

```
  private Phaser phaser;
```

6. Implement the constructor of the class that will initialize the attributes of the class. It receives as parameters the full path of the initial folder, the extension of the files, and the phaser.

```
  public FileSearch(String initPath, String end, Phaser phaser) {
    this.initPath = initPath;
    this.end = end;
    this.phaser=phaser;
    results=new ArrayList<>();
  }
```

7. Now, you have to implement some auxiliary methods that will be used by the `run()` method. The first one is the `directoryProcess()` method. It receives a `File` object as a parameter and it processes all its files and subfolders. For each folder, the method will make a recursive call passing the folder as a parameter. For each file, the method will call the `fileProcess()` method:

```
  private void directoryProcess(File file) {

    File list[] = file.listFiles();
    if (list != null) {
      for (int i = 0; i < list.length; i++) {
```

```
        if (list[i].isDirectory()) {
          directoryProcess(list[i]);
        } else {
          fileProcess(list[i]);
        }
      }
    }
  }
```

8. Now, implement the `fileProcess()` method. It receives a `File` object as parameter and checks if its extension is equal to the one we are looking for. If they are equal, this method adds the absolute path of the file to the list of results.

```
  private void fileProcess(File file) {
    if (file.getName().endsWith(end)) {
      results.add(file.getAbsolutePath());
    }
  }
```

9. Now, implement the `filterResults()` method. It doesn't receive any parameter, and filters the list of files obtained in the first phase, deleting the files that were modified more than 24 hours ago. First, create a new empty list and get the actual date.

```
  private void filterResults() {
    List<String> newResults=new ArrayList<>();
    long actualDate=new Date().getTime();
```

10. Then, go through all the elements of the results list. For each path in the list of results, create a `File` object for that file and get the last modified date for it.

```
    for (int i=0; i<results.size(); i++){
      File file=new File(results.get(i));
      long fileDate=file.lastModified();
```

11. Then, compare that date with the actual date and, if the difference is less than one day, add the full path of the file to the new list of results.

```
      if (actualDate-fileDate< TimeUnit.MILLISECONDS.
convert(1,TimeUnit.DAYS)){
        newResults.add(results.get(i));
      }
    }
```

12. Finally, change the old results list for the new one.

```
    results=newResults;
  }
```

13. Now, implement the `checkResults()` method. This method will be called at the end of the first and the second phase and it will check if the results list is empty or not. This method doesn't have any parameters.

```
  private boolean checkResults() {
```

14. First, check the size of the results list. If it's `0`, the object writes a message to the console indicating this circumstance and then, calls the `arriveAndDeregister()` method of the `Phaser` object to notify it that this thread has finished the actual phase, and it leaves the phased operation.

```
  if (results.isEmpty()) {
      System.out.printf("%s: Phase %d: 0 results.\n",Thread.
currentThread().getName(),phaser.getPhase());
      System.out.printf("%s: Phase %d: End.\n",Thread.
currentThread().getName(),phaser.getPhase());
      phaser.arriveAndDeregister();
      return false;
```

15. Otherwise, if the results list has elements, the object writes a message to the console indicating this circumstance and then, calls the `arriveAndAwaitAdvance()` method of the `Phaser` object to notify it that this thread has finished the actual phase and it wants to be blocked until all the participant threads in the phased operation finish the actual phase.

```
    } else {
    System.out.printf("%s: Phase %d: %d results.\n",Thread.
currentThread().getName(),phaser.getPhase(),results.size());
      phaser.arriveAndAwaitAdvance();
      return true;
    }
  }
```

16. The last auxiliary method is the `showInfo()` method that prints to the console the elements of the results list.

```
  private void showInfo() {
    for (int i=0; i<results.size(); i++){
      File file=new File(results.get(i));
      System.out.printf("%s: %s\n",Thread.currentThread().
getName(),file.getAbsolutePath());
    }
    phaser.arriveAndAwaitAdvance();
  }
```

17. Now, it's time to implement the `run()` method that executes the operation using the auxiliary methods described earlier and the `Phaser` object to control the change between phases. First, call the `arriveAndAwaitAdvance()` method of the `phaser` object. The search won't begin until all the threads have been created.

```
  @Override
  public void run() {

    phaser.arriveAndAwaitAdvance();
```

18. Then, write a message to the console indicating the start of the search task.

```
    System.out.printf("%s: Starting.\n",Thread.currentThread().
getName());
```

19. Check that the `initPath` attribute stores the name of a folder and use the `directoryProcess()` method to find the files with the specified extension in that folder and all its subfolders.

```
    File file = new File(initPath);
    if (file.isDirectory()) {
      directoryProcess(file);
    }
```

20. Check if there are any results using the `checkResults()` method. If there are no results, finish the execution of the thread with the `return` keyword.

```
    if (!checkResults()){
      return;
    }
```

21. Filter the list of results using the `filterResults()` method.

```
    filterResults();
```

22. Check again if there are any results using the `checkResults()` method. If there are no results, finish the execution of the thread with the `return` keyword.

```
    if (!checkResults()){
      return;
    }
```

23. Print the final list of results to the console with the `showInfo()` method, deregister the thread, and print a message indicating the finalization of the thread.

```
    showInfo();
    phaser.arriveAndDeregister();
    System.out.printf("%s: Work completed.\n",Thread.
currentThread().getName());
```

24. Now, implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

25. Create a `Phaser` object with three participants.

```
    Phaser phaser=new Phaser(3);
```

26. Create three `FileSearch` objects with a different initial folder for each one. Look for the files with the `.log` extension.

```
    FileSearch system=new FileSearch("C:\\Windows", "log", 
phaser);
    FileSearch apps=
new FileSearch("C:\\Program Files","log",phaser);
    FileSearch documents=
new FileSearch("C:\\Documents And Settings","log",phaser);
```

27. Create and start a thread to execute the first `FileSearch` object.

```
    Thread systemThread=new Thread(system,"System");
    systemThread.start();
```

28. Create and start a thread to execute the second `FileSearch` object.

```
    Thread appsThread=new Thread(apps,"Apps");
    appsThread.start();
```

29. Create and start a thread to execute the third `FileSearch` object.

```
    Thread documentsThread=new Thread(documents, "Documents");
    documentsThread.start();
```

30. Wait for the finalization of the three threads.

```
    try {
      systemThread.join();
      appsThread.join();
      documentsThread.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

31. Write the value of the finalized flag of the `Phaser` object using the `isFinalized()` method.

```
    System.out.println("Terminated: "+ phaser.isTerminated());
```

## How it works...

The program starts creating a `Phaser` object that will control the synchronization of the threads at the end of each phase. The constructor of `Phaser` receives the number of participants as a parameter. In our case, `Phaser` has three participants. This number indicates to `Phaser` the number of threads that have to execute an `arriveAndAwaitAdvance()` method before `Phaser` changes the phase and wakes up the threads that were sleeping.

Once `Phaser` has been created, we launch three threads that execute three different `FileSearch` objects.

> In this example, we use paths of the Windows operating system. If you work with another operating system, modify the paths to adapt them to existing paths in your environment.

The first instruction in the `run()` method of this `FileSearch` object is a call to the `arriveAndAwaitAdvance()` method of the `Phaser` object. As we mentioned earlier, the `Phaser` knows the number of threads that we want to synchronize. When a thread calls this method, `Phaser` decreases the number of threads that have to finalize the actual phase and puts this thread to sleep until all the remaining threads finish this phase. Calling this method at the beginning of the `run()` method makes none of the `FileSearch` threads begin their job until all the threads have been created.

At the end of phase one and phase two, we check if the phase has generated results and the list with the results has elements, or otherwise the phase hasn't generated results and the list is empty. In the first case, the `checkResults()` method calls `arriveAndAwaitAdvance()` as explained earlier. In the second case, if the list is empty, there's no point in the thread continuing with its execution, so it returns. But you have to notify the phaser that there will be one less participant. For this, we used `arriveAndDeregister()`. This notifies the phaser that this thread has finished the actual phase, but it won't participate in the future phases, so the phaser won't have to wait for it to continue.

At the end of the phase three implemented in the `showInfo()` method, there is a call to the `arriveAndAwaitAdvance()` method of the phaser. With this call, we guarantee that all the threads finish at the same time. When this method ends its execution, there is a call to the `arriveAndDeregister()` method of the phaser. With this call, we deregister the threads of the phaser as we explained before, so when all the threads finish, the phaser will have zero participants.

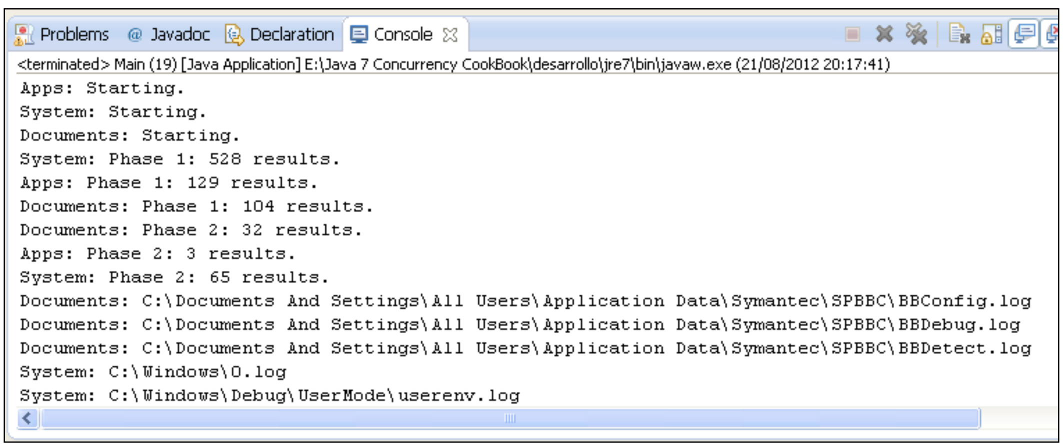
Finally, the `main()` method waits for the completion of the three threads and calls the `isTerminated()` method of the phaser. When a phaser has zero participants, it enters the so called termination state and this method returns `true`. As we deregister all the threads of the phaser, it will be in the termination state and this call will print `true` to the console.

A `Phaser` object can be in two states:

- **Active**: `Phaser` enters this state when it accepts the registration of new participants and its synchronization at the end of each phase. In this state, `Phaser` works as it has been explained in this recipe. This state is not mentioned in the Java concurrency API.
- **Termination**: By default, `Phaser` enters in this state when all the participants in `Phaser` have been deregistered, so `Phaser` has zero participants. More in detail, `Phaser` is in the termination state when the method `onAdvance()` returns the `true` value. If you override that method, you can change the default behavior. When `Phaser` is on this state, the synchronization method `arriveAndAwaitAdvance()` returns immediately without doing any synchronization operation.

A notable feature of the `Phaser` class is that you haven't had to control any exception from the methods related with the phaser. Unlike other synchronization utilities, threads that are sleeping in a phaser don't respond to interruption events and don't throw an `InterruptedException` exception. There is only one exception that is explained in the *There's more* section below.

The following screenshot shows the results of one execution of the example:




It shows the first two phases of the execution. You can see how the **Apps** thread finishes its execution in phase two because its results list is empty. When you execute the example, you will see how some threads finish a phase before the rest, but they wait until all have finished one phase before continuing with the rest.

## There's more...

The `Phaser` class provides other methods related to the change of phase. These methods are as follows:

- `arrive()`: This method notifies the phaser that one participant has finished the actual phase, but it should not wait for the rest of the participants to continue with its execution. Be careful with the utilization of this method, because it doesn't synchronize with other threads.
- `awaitAdvance(int phase)`: This method puts the current thread to sleep until all the participants of the phaser have finished the current phase of the phaser, if the number we pass as the parameter is equal to the actual phase of the phaser. If the parameter and the actual phase of the phaser aren't equal, the method returns immediately.
- `awaitAdvanceInterruptibly(int phaser)`: This method is equal to the method explained earlier, but it throws an `InterruptedException` exception if the thread that is sleeping in this method is interrupted.

## Registering participants in the Phaser

When you create a `Phaser` object, you indicate how many participants will have that phaser. But the `Phaser` class has two methods to increment the number of participants of a phaser. These methods are as follows:

- `register()`: This method adds a new participant to `Phaser`. This new participant will be considered as unarrived to the actual phase.
- `bulkRegister(int Parties)`: This method adds the specified number of participants to the phaser. These new participants will be considered as unarrived to the actual phase.

The only method provided by the `Phaser` class to decrement the number of participants is the `arriveAndDeregister()` method that notifies the phaser that the thread has finished the actual phase, and it doesn't want to continue with the phased operation.

## Forcing the termination of a Phaser

When a phaser has zero participants, it enters a state denoted by **Termination**. The `Phaser` class provides `forceTermination()` to change the status of the phaser and makes it enter in the Termination state independently of the number of participants registered in the phaser. This mechanism may be useful when one of the participants has an error situation, to force the termination of the phaser.

When a phaser is in the Termination state, the `awaitAdvance()` and `arriveAndAwaitAdvance()` methods immediately return a negative number, instead of a positive one that returns normally. If you know that your phaser could be terminated, you should verify the return value of those methods to know if the phaser has been terminated.

## See also

- The *Monitoring a Phaser* recipe in *Chapter 8*, *Testing Concurrent Applications*

# Controlling phase change in concurrent phased tasks

The `Phaser` class provides a method that is executed each time the phaser changes the phase. It's the `onAdvance()` method. It receives two parameters: the number of the current phase and the number of registered participants; it returns a `Boolean` value, `false` if the phaser continues its execution, or `true` if the phaser has finished and has to enter into the termination state.

The default implementation of this method returns `true` if the number of registered participants is zero, and `false` otherwise. But you can modify this behavior if you extend the `Phaser` class and you override this method. Normally, you will be interested in doing this when you have to execute some actions when you advance from one phase to the next one.

In this recipe, you will learn how to control the phase change in a phaser that is implementing your own version of the `Phaser` class that overrides the `onAdvance()` method to execute some actions in every phase change. You are going to implement a simulation of an exam, where there will be some students who have to do three exercises. All the students have to finish one exercise before they can proceed with the next one.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyPhaser` and specify that it extends from the `Phaser` class.

   ```
   public class MyPhaser extends Phaser {
   ```

2. Override the `onAdvance()` method. According to the value of the phase attribute, we call a different auxiliary method. If the phase is equal to zero, you have to call the `studentsArrived()` method. If the phase is equal to one, you have to call the `finishFirstExercise()` method. If the phase is equal to two, you have to call the `finishSecondExercise()` method, and if the phase is equal to three, you have to call the `finishExam()` method. Otherwise, we return the `true` value to indicate that the phaser has terminated.

```
 @Override
protected boolean onAdvance(int phase, int registeredParties) {
  switch (phase) {
  case 0:
    return studentsArrived();
  case 1:
    return finishFirstExercise();
  case 2:
    return finishSecondExercise();
  case 3:
    return finishExam();
  default:
    return true;
  }
}
```

3. Implement the auxiliary method `studentsArrived()`. It writes two log messages to the console and returns the `false` value to indicate that the phaser continues with its execution.

```
  private boolean studentsArrived() {
    System.out.printf("Phaser: The exam are going to start. The 
students are ready.\n");
    System.out.printf("Phaser: We have %d students.\
n",getRegisteredParties());
    return false;
  }
```

4. Implement the auxiliary method `finishFirstExercise()`. It writes two messages to the console and returns the `false` value to indicate that the phaser continues with its execution.

```
  private boolean finishFirstExercise() {
    System.out.printf("Phaser: All the students have finished the 
first exercise.\n");
    System.out.printf("Phaser: It's time for the second one.\n");
    return false;
  }
```

5. Implement the auxiliary method `finishSecondExercise()`. It writes two messages to the console and returns the `false` value to indicate that the phaser continues with its execution.

```
  private boolean finishSecondExercise() {
    System.out.printf("Phaser: All the students have finished the 
second exercise.\n");
    System.out.printf("Phaser: It's time for the third one.\n");
    return false;
  }
```

6. Implement the auxiliary method `finishExam()`. It writes two messages to the console and returns the `true` value to indicate that the phaser has finished its work.

```
  private boolean finishExam() {
    System.out.printf("Phaser: All the students have finished the 
exam.\n");
    System.out.printf("Phaser: Thank you for your time.\n");
    return true;
  }
```

7. Create a class named `Student` and specify that it implements the `Runnable` interface. This class will simulate the students of the exam.

```
public class Student implements Runnable {
```

8. Declare a `Phaser` object named `phaser`.

```
  private Phaser phaser;
```

9. Implement the constructor of the class that initializes the `Phaser` object.

```
  public Student(Phaser phaser) {
    this.phaser=phaser;
  }
```

10. Implement the `run()` method that will simulate the realization of the exam.

```
   @Override
  public void run() {
```

11. First, the method writes a message in the console to indicate that this student has arrived to the exam and calls the `arriveAndAwaitAdvance()` method of the phaser to wait for the rest of the threads.

```
    System.out.printf("%s: Has arrived to do the exam. 
%s\n",Thread.currentThread().getName(),new Date());
    phaser.arriveAndAwaitAdvance();
```

12. Then, write a message to the console, call the private `doExercise1()` method that simulates the realization of the first exercise of the exam, write another message to the console and the `arriveAndAwaitAdvance()` method of the phaser to wait for the rest of the students to finish the first exercise.

```
    System.out.printf("%s: Is going to do the first exercise. 
%s\n",Thread.currentThread().getName(),new Date());
    doExercise1();
    System.out.printf("%s: Has done the first exercise. 
%s\n",Thread.currentThread().getName(),new Date());
    phaser.arriveAndAwaitAdvance();
```

13. Implement the same code for second exercise and third execise.

```
    System.out.printf("%s: Is going to do the second exercise. 
```

```
%s\n",Thread.currentThread().getName(),new Date());
    doExercise2();
    System.out.printf("%s: Has done the second exercise.
%s\n",Thread.currentThread().getName(),new Date());
    phaser.arriveAndAwaitAdvance();
    System.out.printf("%s: Is going to do the third exercise.
%s\n",Thread.currentThread().getName(),new Date());
    doExercise3();
    System.out.printf("%s: Has finished the exam. %s\n",Thread.
currentThread().getName(),new Date());
    phaser.arriveAndAwaitAdvance();
```

14. Implement the auxiliary method `doExercise1()`. This method puts the thread to sleep for a random period of time.

```
private void doExercise1() {
  try {
    long duration=(long)(Math.random()*10);
    TimeUnit.SECONDS.sleep(duration);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

15. Implement the auxiliary method `doExercise2()`. This method puts the thread to sleep for a random period of time.

```
private void doExercise2() {
  try {
    long duration=(long)(Math.random()*10);
    TimeUnit.SECONDS.sleep(duration);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

16. Implement the auxiliary method `doExercise3()`. This method puts the thread to sleep for a random period of time.

```
private void doExercise3() {
  try {
    long duration=(long)(Math.random()*10);
    TimeUnit.SECONDS.sleep(duration);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

17. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

18. Create a `MyPhaser` object.

```
    MyPhaser phaser=new MyPhaser();
```

19. Create five `Student` objects and register them in the phaser using the `register()` method.

```
    Student students[]=new Student[5];
    for (int i=0; i<students.length; i++){
      students[i]=new Student(phaser);
      phaser.register();
    }
```

20. Create five threads to run `students` and start them.

```
    Thread threads[]=new Thread[students.length];
    for (int i=0; i<students.length; i++){
      threads[i]=new Thread(students[i],"Student "+i);
      threads[i].start();
    }
```

21. Wait for the finalization of the five threads.

```
    for (int i=0; i<threads.length; i++){
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

22. Write a message to show that the phaser is in the termination state using the `isTerminated()` method.

```
    System.out.printf("Main: The phaser has finished: %s.\
n",phaser.isTerminated());
```

## How it works...

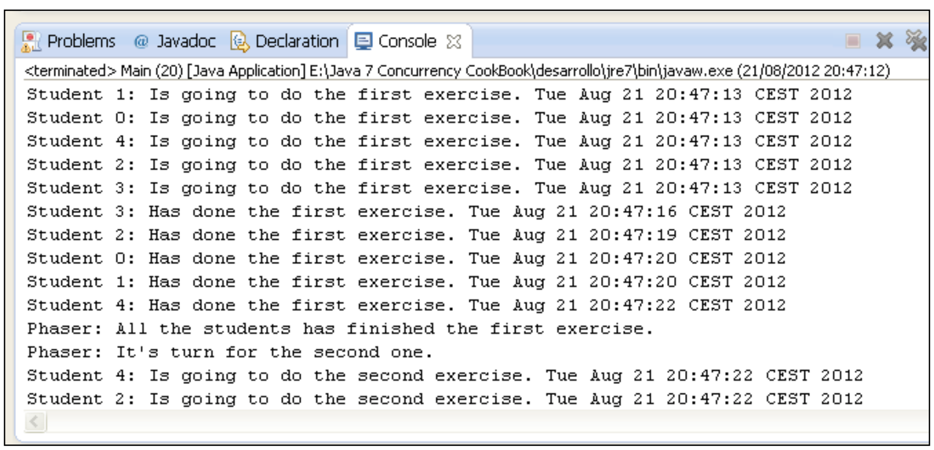
This exercise simulates the realization of an exam that has three exercises. All the students have to finish one exercise before they can start the next one. To implement this synchronization requirement, we use the `Phaser` class, but you have implemented your own phaser extending the original class to override the `onAdvance()` method.

This method is called by the phaser before making a phase change and before waking up all the threads that were sleeping in the `arriveAndAwaitAdvance()` method. This method receives as parameters the number of the actual phase, where `0` is the number of the first phase and the number of registered participants. The most useful parameter is the actual phase. If you execute a different operation depending on the actual phase, you have to use an alternative structure (`if/else` or `switch`) to select the operation you want to execute. In the example, we used a `switch` structure to select a different method for each change of phase.

The `onAdvance()` method returns a `Boolean` value that indicates if the phaser has terminated or not. If the phaser returns a `false` value, it indicates that it hasn't terminated, so the threads will continue with the execution of other phases. If the phaser returns a `true` value, then the phaser still wakes up the pending threads, but moves the phaser to the terminated state, so all the future calls to any method of the phaser will return immediately, and the `isTerminated()` method returns the `true` value.

In the `Core` class, when you created the `MyPhaser` object, you didn't specify the number of participants in the phaser. You made a call to the `register()` method for every `Student` object created to register a participant in the phaser. This calling doesn't establish a relation between the `Student` object or the thread that executes it and the phaser. Really, the number of participants in a phaser is only a number. There is no relationship between the phaser and the participants.

The following screenshot shows the results of an execution of this example:




You can see how the students finish the first exercise at different times. When all have finished that exercise, the phaser calls the `onAdvance()` method that writes the log messages in the console and then all the students start the second exercise at the same time.

## See also

- The *Running concurrent phased tasks* recipe in *Chapter 3*, *Thread Synchronization Utilities*
- The *Monitoring a Phaser* recipe in *Chapter 8*, *Testing Concurrent Applications*

# Changing data between concurrent tasks

The Java concurrency API provides a synchronization utility that allows the interchange of data between two concurrent tasks. In more detail, the `Exchanger` class allows the definition of a synchronization point between two threads. When the two threads arrive to this point, they interchange a data structure so the data structure of the first thread goes to the second one and the data structure of the second thread goes to the first one.

This class may be very useful in a situation similar to the producer-consumer problem. This is a classic concurrent problem where you have a common buffer of data, one or more producers of data, and one or more consumers of data. As the `Exchanger` class only synchronizes two threads, you can use it if you have a producer-consumer problem with one producer and one consumer.

In this recipe, you will learn how to use the `Exchanger` class to solve the producer-consumer problem with one producer and one consumer.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE like NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. First, let's begin by implementing the producer. Create a class named `Producer` and specify that it implements the `Runnable` interface.

   ```
   public class Producer implements Runnable {
   ```

2. Declare a `List<String>` object named `buffer`. This will be the data structure that the producer will interchange with the consumer.

   ```
     private List<String> buffer;
   ```

3. Declare an `Exchanger<List<String>>` object named `exchanger`. This will be the exchanger object that will be used to synchronize producer and consumer.

   ```
     private final Exchanger<List<String>> exchanger;
   ```

4. Implement the constructor of the class that initializes the two attributes.

```
  public Producer (List<String> buffer, Exchanger<List<String>>
exchanger){
    this.buffer=buffer;
    this.exchanger=exchanger;
  }
```

5. Implement the `run()` method. Inside it, implement 10 cycles of interchange.

```
  @Override
  public void run() {
    int cycle=1;

    for (int i=0; i<10; i++){
      System.out.printf("Producer: Cycle %d\n",cycle);
```

6. In each cycle, add 10 strings to the buffer.

```
      for (int j=0; j<10; j++){
        String message="Event "+((i*10)+j);
        System.out.printf("Producer: %s\n",message);
        buffer.add(message);
      }
```

7. Call the `exchange()` method to interchange data with the consumer. As this method can throw an `InterruptedException` exception, you have to add the code to process it.

```
      try {
        buffer=exchanger.exchange(buffer);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      System.out.println("Producer: "+buffer.size());
      cycle++;
    }
```

8. Now, let's implement the consumer. Create a class named `Consumer` and specify that it implements the `Runnable` interface.

```
public class Consumer implements Runnable {
```

9. Declare a `List<String>` object named `buffer`. This will be the data structure that the producer will interchange with the consumer.

```
  private List<String> buffer;
```

10. Declare an `Exchanger<List<String>>` object named `exchanger`. This will be the exchanger object that will be used to synchronize producer and consumer.

```
  private final Exchanger<List<String>> exchanger;
```

11. Implement the constructor of the class that initializes the two attributes.

```
  public Consumer(List<String> buffer, Exchanger<List<String>>
exchanger){
    this.buffer=buffer;
    this.exchanger=exchanger;
  }
```

12. Implement the `run()` method. Inside it, implement 10 cycles of interchange.

```
  @Override
  public void run() {
    int cycle=1;

    for (int i=0; i<10; i++){
      System.out.printf("Consumer: Cycle %d\n",cycle);
```

13. In each cycle, begin with a call to the `exchange()` method to synchronize with the producer. The consumer needs data to consume. As this method can throw an `InterruptedException` exception, you have to add the code to process it.

```
      try {
        buffer=exchanger.exchange(buffer);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
```

14. Write the 10 strings the producer sent in its buffer to the console and delete them from the buffer, to leave it empty.

```
      System.out.println("Consumer: "+buffer.size());

      for (int j=0; j<10; j++){
        String message=buffer.get(0);
        System.out.println("Consumer: "+message);
        buffer.remove(0);
      }

      cycle++;
    }
```

15. Now, implement the main class of the example by creating a class named `Core` and add the `main()` method to it.

```
public class Core {

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

16. Create the two buffers that will be used by the producer and the consumer.

```
    List<String> buffer1=new ArrayList<>();
    List<String> buffer2=new ArrayList<>();
```

17. Create the `Exchanger` object that will be used to synchronize the producer and the consumer.

```
    Exchanger<List<String>> exchanger=new Exchanger<>();
```

18. Create the `Producer` object and the `Consumer` object.

```
    Producer producer=new Producer(buffer1, exchanger);
    Consumer consumer=new Consumer(buffer2, exchanger);
```

19. Create the threads to execute the producer and the consumer and start the threads.

```
    Thread threadProducer=new Thread(producer);
    Thread threadConsumer=new Thread(consumer);

    threadProducer.start();
    threadConsumer.start();
```

## How it works...

The consumer begins with an empty buffer and calls `Exchanger` to synchronize with the producer. It needs data to consume. The producer begins its execution with an empty buffer. It creates 10 strings, stores it in the buffer, and uses the exchanger to synchronize with the consumer.

At this point, both threads (producer and consumer) are in `Exchanger` and it changes the data structures, so when the consumer returns from the `exchange()` method, it will have a buffer with 10 strings. When the producer returns from the `exchange()` method, it will have an empty buffer to fill again. This operation will be repeated 10 times.

If you execute the example, you will see how producer and consumer do their jobs concurrently and how the two objects interchange their buffers in every step. As it occurs with other synchronization utilities, the first thread that calls the `exchange()` method was put to sleep until the other threads arrived.

## There's more...

The `Exchanger` class has another version of the exchange method: `exchange(V data, long time, TimeUnit unit)` where `V` is the type used as a parameter in the declaration of `Phaser` (`List<String>` in our case). The thread will be sleeping until it's interrupted, the other thread arrives, or the specified time passes. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

# 4
# Thread Executors

In this chapter, we will cover:

- Creating a thread executor
- Creating a fixed-size thread executor
- Executing tasks in an executor that returns a result
- Running multiple tasks and processing the first result
- Running multiple tasks and processing all the results
- Running a task in an executor after a delay
- Running a task in an executor periodically
- Canceling a task in an executor
- Controlling a task finishing in an executor
- Separating the launching of tasks and the processing of their results in an executor
- Controlling the rejected tasks of an executor

## Introduction

Usually, when you develop a simple, concurrent-programming application in Java, you create some `Runnable` objects and then create the corresponding `Thread` objects to execute them. If you have to develop a program that runs a lot of concurrent tasks, this approach has the following disadvantages:

- You have to implement all the code-related information to the management of the `Thread` objects (creation, ending, obtaining results).
- You create a `Thread` object per task. If you have to execute a big number of tasks, this can affect the throughput of the application.

- You have to control and manage efficiently the resources of the computer. If you create too many threads, you can saturate the system.

Since Java 5, the Java concurrency API provides a mechanism that aims at resolving problems. This mechanism is called the **Executor framework** and is around the `Executor` interface, its subinterface `ExecutorService`, and the `ThreadPoolExecutor` class that implements both interfaces.

This mechanism separates the task creation and its execution. With an executor, you only have to implement the `Runnable` objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads. But it goes beyond that and improves performance using a pool of threads. When you send a task to the executor, it tries to use a pooled thread for the execution of this task, to avoid continuous spawning of threads.

Another important advantage of the Executor framework is the `Callable` interface. It's similar to the `Runnable` interface, but offers two improvements, which are as follows:

- The main method of this interface, named `call()`, may return a result.
- When you send a `Callable` object to an executor, you get an object that implements the `Future` interface. You can use this object to control the status and the result of the `Callable` object.

This chapter presents 11 recipes that show you how to work with the Executor framework using the classes mentioned earlier and other variants provided by the Java Concurrency API.

# Creating a thread executor

The first step to work with the Executor framework is to create an object of the `ThreadPoolExecutor` class. You can use the four constructors provided by that class or use a factory class named `Executors` that creates `ThreadPoolExecutor`. Once you have an executor, you can send `Runnable` or `Callable` objects to be executed.

In this recipe, you will learn how these two operations implement an example that will simulate a web server processing requests from various clients.

## Getting ready

You should read the *Creating and running a thread* recipe in *Chapter 1* to learn the basic mechanism of thread creation in Java. You can compare both mechanisms and select the best one depending on the problem.

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. First, you have to implement the tasks that will be executed by the server. Create a class named `Task` that implements the `Runnable` interface.

```
public class Task implements Runnable {
```

2. Declare a `Date` attribute named `initDate` to store the creation date of the task and a `String` attribute named `name` to store the name of the task.

```
  private Date initDate;
  private String name;
```

3. Implement the constructor of the class that initializes both attributes.

```
  public Task(String name){
    initDate=new Date();
    this.name=name;
  }
```

4. Implement the `run()` method.

```
   @Override
  public void run() {
```

5. First, write to the console the `initDate` attribute and the actual date, which is the starting date of the task.

```
    System.out.printf("%s: Task %s: Created on: %s\n",Thread.
currentThread().getName(),name,initDate);
    System.out.printf("%s: Task %s: Started on: %s\n",Thread.
currentThread().getName(),name,new Date());
```

6. Then, put the task to sleep for a random period of time.

```
    try {
      Long duration=(long)(Math.random()*10);
      System.out.printf("%s: Task %s: Doing a task during %d
seconds\n",Thread.currentThread().getName(),name,duration);
      TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

7. Finally, write to the console the completion date of the task.

```
    System.out.printf("%s: Task %s: Finished on: %s\n",Thread.
currentThread().getName(),name,new Date());
```

8. Now, implement the `Server` class that will execute every task it receives using an executor. Create a class named `Server`.

```
public class Server {
```

9. Declare a `ThreadPoolExecutor` attribute named `executor`.

```
  private ThreadPoolExecutor executor;
```

10. Implement the constructor of the class that initializes the `ThreadPoolExecutor` object using the `Executors` class.

```
  public Server(){
  executor=(ThreadPoolExecutor)Executors.newCachedThreadPool();
  }
```

11. Implement the `executeTask()` method. It receives a `Task` object as a parameter and sends it to the executor. First, write a message to the console indicating that a new task has arrived.

```
  public void executeTask(Task task){
    System.out.printf("Server: A new task has arrived\n");
```

12. Then, call the `execute()` method of the executor to send it the task.

```
    executor.execute(task);
```

13. Finally, write some executor data to the console to see its status.

```
    System.out.printf("Server: Pool Size: %d\n",executor.
getPoolSize());
    System.out.printf("Server: Active Count: %d\n",executor.
getActiveCount());
    System.out.printf("Server: Completed Tasks: %d\n",executor.
getCompletedTaskCount());
```

14. Implement the `endServer()` method. In this method, call the `shutdown()` method of the executor to finish its execution.

```
  public void endServer() {
    executor.shutdown();
  }
```

15. Finally, implement the main class of the example by creating a class named `Main` and implement the `main()` method.

```
public class Main {

  public static void main(String[] args) {
    Server server=new Server();
    for (int i=0; i<100; i++){
      Task task=new Task("Task "+i);
```

```
            server.executeTask(task);
          }
          server.endServer();
        }
      }
```

## How it works...

The key of this example is the `Server` class. This class creates and uses `ThreadPoolExecutor` to execute tasks.

The first important point is the creation of `ThreadPoolExecutor` in the constructor of the `Server` class. The `ThreadPoolExecutor` class has four different constructors but, due to their complexity, the Java concurrency API provides the `Executors` class to construct executors and other related objects. Although we can create `ThreadPoolExecutor` directly using one of its constructors, it's recommended to use the `Executors` class.

In this case, you have created a cached thread pool using the `newCachedThreadPool()` method. This method returns an `ExecutorService` object, so it's been cast to `ThreadPoolExecutor` to have access to all its methods. The cached thread pool you have created creates new threads if needed to execute the new tasks, and reuses the existing ones if they have finished the execution of the task they were running, which are now available. The reutilization of threads has the advantage that it reduces the time taken for thread creation. The cached thread pool has, however, a disadvantage of constant lying threads for new tasks, so if you send too many tasks to this executor, you can overload the system.

> Use the executor created by the `newCachedThreadPool()` method only when you have a reasonable number of threads or when they have a short duration.

Once you have created the executor, you can send tasks of the `Runnable` or `Callable` type for execution using the `execute()` method. In this case, you send objects of the `Task` class that implements the `Runnable` interface.

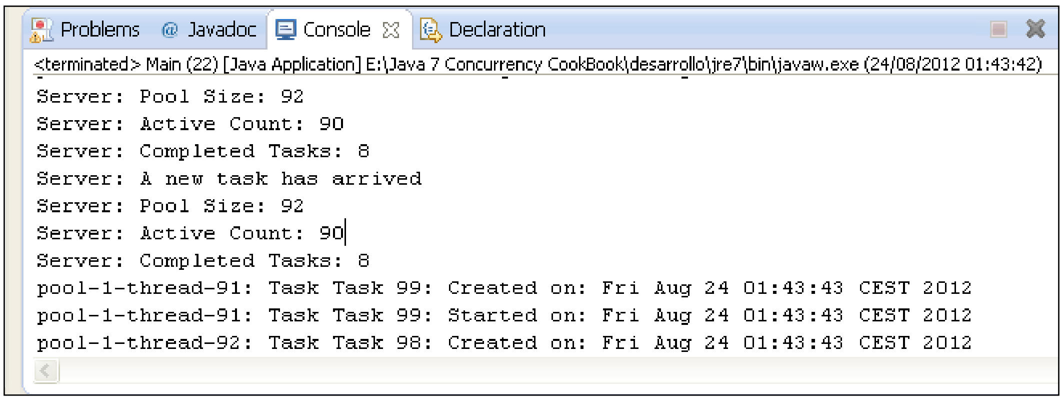
You also have printed some log messages with information about the executor. Specifically, you have used the following methods:

- `getPoolSize()`: This method returns the actual number of threads in the pool of the executor
- `getActiveCount()`: This method returns the number of threads that are executing tasks in the executor
- `getCompletedTaskCount()`: This method returns the number of tasks completed by the executor

One critical aspect of the `ThreadPoolExecutor` class, and of the executors in general, is that you have to end it explicitly. If you don't do this, the executor will continue its execution and the program won't end. If the executor doesn't have tasks to execute, it continues waiting for new tasks and it doesn't end its execution. A Java application won't end until all its non-daemon threads finish their execution, so, if you don't terminate the executor, your application will never end.

To indicate to the executor that you want to finish it, you can use the `shutdown()` method of the `ThreadPoolExecutor` class. When the executor finishes the execution of all pending tasks, it finishes its execution. After you call the `shutdown()` method, if you try to send another task to the executor, it will be rejected and the executor will throw a `RejectedExecutionException` exception.

The following screenshot shows part of one execution of this example:




When the last task arrives to the server, the executor has a pool of 100 tasks and 97 active threads.

## There's more...

The `ThreadPoolExecutor` class provides a lot of methods to obtain information about its status. We used in the example the `getPoolSize()`, `getActiveCount()`, and `getCompletedTaskCount()` methods to obtain information about the size of the pool, the number of threads, and the number of completed tasks of the executor. You can also use the `getLargestPoolSize()` method that returns the maximum number of threads that has been in the pool at a time.

The `ThreadPoolExecutor` class also provides other methods related with the finalization of the executor. These methods are:

- `shutdownNow()`: This method shut downs the executor immediately. It doesn't execute the pending tasks. It returns a list with all these pending tasks. The tasks that are running when you call this method continue with their execution, but the method doesn't wait for their finalization.
- `isTerminated()`: This method returns `true` if you have called the `shutdown()` or `shutdownNow()` methods and the executor finishes the process of shutting it down.
- `isShutdown()`: This method returns `true` if you have called the `shutdown()` method of the executor.
- `awaitTermination(long timeout, TimeUnit unit)`: This method blocks the calling thread until the tasks of the executor have ended or the timeout occurs. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

> If you want to wait for the completion of the tasks, regardless of their duration, use a big timeout, for example, `DAYS`.

## See also

- The *Controlling rejected tasks of an executor* recipe in *Chapter 4*, *Thread Executors*
- The *Monitoring an Executor framework* recipe in *Chapter 8*, *Testing Concurrent Applications*

# Creating a fixed-size thread executor

When you use basic `ThreadPoolExecutor` created with the `newCachedThreadPool()` method of the `Executors` class, you can have a problem with the number of threads the executor is running at a time. The executor creates a new thread for each task that receives, (if there is no pooled thread free) so, if you send a large number of tasks and they have long duration, you can overload the system and provoke a poor performance of your application.

If you want to avoid this problem, the `Executors` class provides a method to create a fixed-size thread executor. This executor has a maximum number of threads. If you send more tasks than the number of threads, the executor won't create additional threads and the remaining tasks will be blocked until the executor has a free thread. With this behavior, you guarantee that the executor won't yield a poor performance of your application.

In this recipe, you are going to learn how to create a fixed-size thread executor modifying the example implemented in the first recipe of this chapter.

## Getting ready

You should read the *Creating a thread executor* recipe in this chapter and implement the example explained in it, because you're going to modify this example.

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Implement the example described in the first recipe of this chapter. Open the `Server` class and modify its constructor. Use the `newFixedThreadPool()` method to create the executor and pass the number `5` as the parameter.

```
public Server(){
executor=(ThreadPoolExecutor)Executors.newFixedThreadPool(5);
}
```

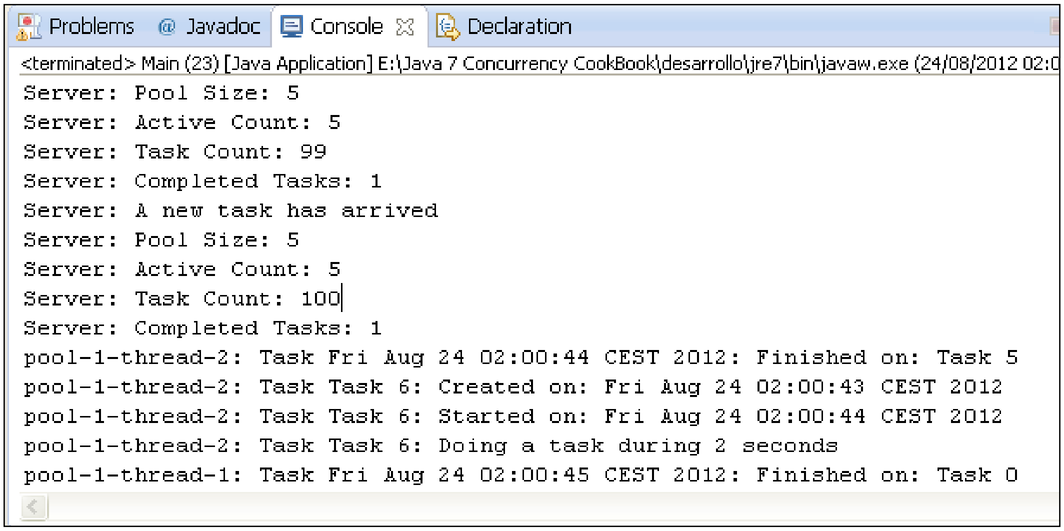
2. Modify the `executeTask()` method including an additional line of log message. Call the `getTaskCount()` method to obtain the number of tasks that have been sent to the executor.

```
    System.out.printf("Server: Task Count: %d\n",executor.
getTaskCount());
```

## How it works...

In this case, you have used the `newFixedThreadPool()` method of the `Executors` class to create the executor. This method creates an executor with a maximum number of threads. If you send more tasks than the number of threads, the remaining tasks will be blocked until there is a free thread to process them This method receives the maximum number of threads as a parameter you want to have in your executor. In your case, you have created an executor with five threads.

The following screenshot shows part of the output of one execution of this example:




To write the output of the program, you have used some methods of the `ThreadPoolExecutor` class, including:

- `getPoolSize()`: This method returns the actual number of threads in the pool of the executor
- `getActiveCount()`: This method returns the number of threads that are executing tasks in the executor

You can see how the output of these methods is **5**, indicating that the executor has five threads. It does not exceed the established maximum number of threads.

When you send the last task to the executor, it has only **5** active threads. The remaining 95 tasks are waiting for free threads. We used the `getTaskCount()` method to show how many you have sent to the executor.

## There's more...

The `Executors` class also provides the `newSingleThreadExecutor()` method. This is an extreme case of a fixed-size thread executor. It creates an executor with only one thread, so it can only execute one task at a time.

## See also

- The *Creating a thread executor* recipe in *Chapter 4*, *Thread Executors*
- The *Monitoring an Executor framework* recipe in *Chapter 8*, *Testing Concurrent Applications*

# Executing tasks in an executor that returns a result

One of the advantages of the Executor framework is that you can run concurrent tasks that return a result. The Java Concurrency API achieves this with the following two interfaces:

- `Callable`: This interface has the `call()` method. In this method, you have to implement the logic of a task. The `Callable` interface is a parameterized interface, meaning you have to indicate the type of data the `call()` method will return.
- `Future`: This interface has some methods to obtain the result generated by a `Callable` object and to manage its state.

In this recipe, you will learn how to implement tasks that return a result and run them on an executor.

## Getting ready...

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `FactorialCalculator`. Specify that it implements the `Callable` interface parameterized with the `Integer` type.

```
public class FactorialCalculator implements Callable<Integer> {
```

2. `Declare` a private `Integer` attribute called `number` to store the number that this task will use for its calculations.

```
  private Integer number;
```

3. Implement the constructor of the class that initializes the attribute of the class.

```
  public FactorialCalculator(Integer number){
    this.number=number;
  }
```

4. Implement the `call()` method. This method returns the factorial of the `number` attribute of `FactorialCalculator`.

```
   @Override
  public Integer call() throws Exception {
```

5. First, create and initialize the internal variables used in the method.

```
int result = 1;
```

6. If the number is `0` or `1`, return `1`. Otherwise, calculate the factorial of the number. Between two multiplications, and for educational purposes, put this task to sleep for 20 milliseconds.

```
if ((num==0)||(num==1)) {
  result=1;
} else {
  for (int i=2; i<=number; i++) {
    result*=i;
    TimeUnit.MILLISECONDS.sleep(20);
  }
}
```

7. Write a message to the console with the result of the operation.

```
    System.out.printf("%s: %d\n",Thread.currentThread().
getName(),result);
```

8. Return the result of the operation.

```
return result;
```

9. Implement the main class of the example by creating a class named `Main` and implement the `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

10. Create `ThreadPoolExecutor` to run the tasks using the `newFixedThreadPool()` method of the `Executors` class. Pass `2` as the parameter.

```
    ThreadPoolExecutor executor=(ThreadPoolExecutor)Executors.
newFixedThreadPool(2);
```

11. Create a list of `Future<Integer>` objects.

```
List<Future<Integer>> resultList=new ArrayList<>();
```

12. Create a random number generator with the `Random` class.

```
Random random=new Random();
```

13. Generate 10 new random integers between zero and 10.

```
for (int i=0; i<10; i++){
  Integer number= random.nextInt(10);
```

14. Create a `FactorialCaculator` object passing this random number as a parameter.

```
        FactorialCalculator calculator=new
FactorialCalculator(number);
```

15. Call the `submit()` method of the executor to send the `FactorialCalculator` task to the executor. This method returns a `Future<Integer>` object to manage the task, and eventually get its result.

```
        Future<Integer> result=executor.submit(calculator);
```

16. Add the `Future` object to the list created before.

```
        resultList.add(result);
      }
```

17. Create a `do` loop to monitor the status of the executor.

```
      do {
```

18. First, write a message to the console indicating the number of completed tasks with the `getCompletedTaskNumber()` method of the executor.

```
        System.out.printf("Main: Number of Completed Tasks:
%d\n",executor.getCompletedTaskCount());
```

19. Then, for the 10 `Future` objects in the list, write a message indicating whether the tasks that it manages have finished or not using the `isDone()` method.

```
        for (int i=0; i<resultList.size(); i++) {
          Future<Integer> result=resultList.get(i);
          System.out.printf("Main: Task %d: %s\n",i,result.
isDone());
        }
```

20. Put the thread to sleep for 50 milliseconds.

```
        try {
          TimeUnit.MILLISECONDS.sleep(50);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
```

21. Repeat this loop while the number of completed tasks of the executor is less than 10.

```
      } while (executor.getCompletedTaskCount()<resultList.size());
```

22. Write to the console the results obtained by each task. For each `Future` object, get the `Integer` object returned by its task using the `get()` method.

```
      System.out.printf("Main: Results\n");
      for (int i=0; i<resultList.size(); i++) {
        Future<Integer> result=resultList.get(i);
```

```
        Integer number=null;
        try {
          number=result.get();
        } catch (InterruptedException e) {
          e.printStackTrace();
        } catch (ExecutionException e) {
          e.printStackTrace();
        }
```

23. Then, print the number to the console.

```
        System.out.printf("Main: Task %d: %d\n",i,number);
      }
```

24. Finally, call the `shutdown()` method of the executor to finalize its execution.

```
      executor.shutdown();
```

## How it works...

In this recipe, you have learned how to use the `Callable` interface to launch concurrent tasks that return a result. You have implemented the `FactorialCalculator` class that implements the `Callable` interface with `Integer` as the type of the result. Hence, it returns before type of the `call()` method.

The other critical point of this example is in the `Main` class. You send a `Callable` object to be executed in an executor using the `submit()` method. This method receives a `Callable` object as a parameter and returns a `Future` object that you can use with two main objectives:

- You can control the status of the task: you can cancel the task and check if it has finished. For this purpose, you have used the `isDone()` method to check if the tasks had finished.
- You can get the result returned by the `call()` method. For this purpose, you have used the `get()` method. This method waits until the `Callable` object has finished the execution of the `call()` method and has returned its result. If the thread is interrupted while the `get()` method is waiting for the result, it throws an `InterruptedException` exception. If the `call()` method throws an exception, this method throws an `ExecutionException` exception.

## There's more...

When you call the `get()` method of a `Future` object and the task controlled by this object hasn't finished yet, the method blocks until the task finishes. The `Future` interface provides another version of the `get()` method.

- `get(long timeout, TimeUnit unit)`: This version of the get method, if the result of the task isn't available, waits for it for the specified time. If the specified period of time passes and the result isn't yet available, the method returns a `null` value. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

## See also

- The *Creating a thread executor* recipe in *Chapter 4*, *Thread Executors*
- The *Running multiple tasks and processing the first result* recipe in *Chapter 4*, *Thread Executors*
- The *Running multiple tasks and processing all the results* recipe in *Chapter 4*, *Thread Executors*

# Running multiple tasks and processing the first result

A common problem in concurrent programming is when you have various concurrent tasks that solve a problem, and you are only interested in the first result of those tasks. For example, you want to sort an array. You have various sort algorithms. You can launch all of them and get the result of the first one that sorts these, that is, the fastest sorting algorithm for a given array.

In this recipe, you will learn how to implement this scenario using the `ThreadPoolExecutor` class. You are going to implement an example where a user can be validated by two mechanisms. The user will be validated if one of those mechanisms validates it.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `UserValidator` that will implement the process of user validation.

   ```
   public class UserValidator {
   ```

2. Declare a private `String` attribute named `name` that will store the name of a user validation system.

   ```
     private String name;
   ```

3. Implement the constructor of the class that initializes its attributes.

   ```
     public UserValidator(String name) {
       this.name=name;
     }
   ```

4. Implement the `validate()` method. It receives two `String` parameters with the name and the password of the user you want to validate.

   ```
     public boolean validate(String name, String password) {
   ```

5. Create a `Random` object named `random`.

   ```
       Random random=new Random();
   ```

6. Wait for a random period of time to simulate the process of user validation.

   ```
       try {
         long duration=(long)(Math.random()*10);
         System.out.printf("Validator %s: Validating a user during %d
   seconds\n",this.name,duration);
         TimeUnit.SECONDS.sleep(duration);
       } catch (InterruptedException e) {
         return false;
       }
   ```

7. Return a random `Boolean` value. The method returns a `true` value when the user is validated and a `false` value when the user is not validated.

   ```
       return random.nextBoolean();
     }
   ```

8. Implement the `getName()` method. This method returns the value of the name attribute.

   ```
     public String getName(){
       return name;
     }
   ```

9. Now, create a class named `TaskValidator` that will execute a validation process with a `UserValidation` object as a concurrent task. Specify that it implements the `Callable` interface parameterized with the `String` class.

```
public class TaskValidator implements Callable<String> {
```

10. Declare a private `UserValidator` attribute named `validator`.

```
  private UserValidator validator;
```

11. Declare two private `String` attributes named `user` and `password`.

```
  private String user;
  private String password;
```

12. Implement the constructor of the class that will initialize all the attributes.

```
  public TaskValidator(UserValidator validator, String user, 
String password){
    this.validator=validator;
    this.user=user;
    this.password=password;
  }
```

13. Implement the `call()` method that will return a `String` object.

```
  @Override
  public String call() throws Exception {
```

14. If the user is not validated by the `UserValidator` object, write a message to the console indicating this circumstance and throw an `Exception` exception.

```
    if (!validator.validate(user, password)) {
      System.out.printf("%s: The user has not been found\
n",validator.getName());
      throw new Exception("Error validating user");
    }
```

15. Otherwise, write a message to the console indicating that the user has been validated and return the name of the `UserValidator` object.

```
    System.out.printf("%s: The user has been found\n",validator.
getName());
    return validator.getName();
```

16. Now, implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

17. Create two `String` objects named `user` and `password` and initialize them with the `test` value.

```
String username="test";
String password="test";
```

18. Create two `UserValidator` objects named `ldapValidator` and `dbValidator`.

```
UserValidator ldapValidator=new UserValidator("LDAP");
UserValidator dbValidator=new UserValidator("DataBase");
```

19. Create two `TaskValidator` objects named `ldapTask` and `dbTask`. Initialize them with `ldapValidator` and `dbValidator` respectively.

```
    TaskValidator ldapTask=new TaskValidator(ldapValidator,
username, password);
    TaskValidator dbTask=new TaskValidator(dbValidator,username,pa
ssword);
```

20. Create a list of `TaskValidator` objects and add to it the two objects that you have created.

```
List<TaskValidator> taskList=new ArrayList<>();
taskList.add(ldapTask);
taskList.add(dbTask);
```

21. Create a new `ThreadPoolExecutor` object using the `newCachedThreadPool()` method of the `Executors` class and a `String` object named `result`.

```
    ExecutorService executor=(ExecutorService)Executors.
newCachedThreadPool();
    String result;
```

22. Call the `invokeAny()` method of the `executor` object. This method receives `taskList` as a parameter and returns `String`. Also, it writes the `String` object returned by this method to the console.

```
try {
  result = executor.invokeAny(taskList);
  System.out.printf("Main: Result: %s\n",result);
} catch (InterruptedException e) {
  e.printStackTrace();
} catch (ExecutionException e) {
  e.printStackTrace();
}
```

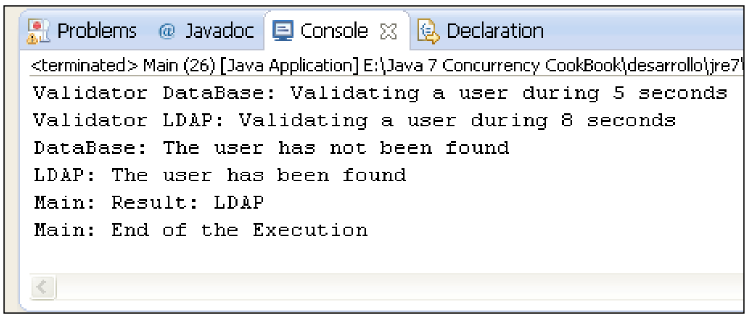
23. Terminate the executor using the `shutdown()` method and write a message to the console to indicate that the program has ended.

```
executor.shutdown();
System.out.printf("Main: End of the Execution\n");
```

## How it works...

The key of the example is in the `Main` class. The `invokeAny()` method of the `ThreadPoolExecutor` class receives a list of tasks, launches them, and returns the result of the first task that finishes without throwing an exception. This method returns the same data type that the `call()` method of the tasks you launch returns. In this case, it returns a `String` value.

The following screenshot shows the output of an execution of the example when one task validates the user:



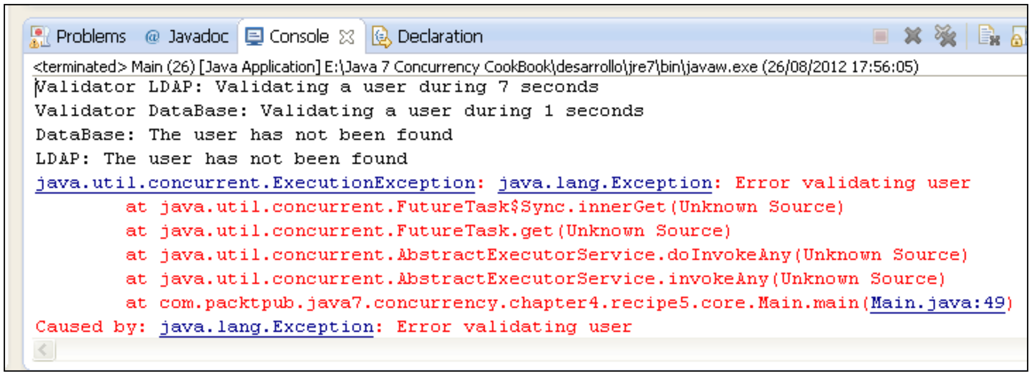

The example has two `UserValidator` objects that return a random `boolean` value. Each `UserValidator` object is used by a `Callable` object, implemented by the `TaskValidator` class. If the `validate()` method of the `UserValidator` class returns a `false` value, the `TaskValidator` class throws `Exception`. Otherwise, it returns the `true` value.

So, we have two tasks that can return the `true` value or throw an `Exception` exception. You can have the following four possibilities:

- Both tasks return the `true` value. The result of the `invokeAny()` method is the name of the task that finishes in the first place.
- The first task returns the `true` value and the second one throws `Exception`. The result of the `invokeAny()` method is the name of the first task.
- The first task throws `Exception` and the second one returns the `true` value. The result of the `invokeAny()` method is the name of the second task.
- Both tasks throw `Exception`. In that class, the `invokeAny()` method throws an `ExecutionException` exception.

If you run the examples several times, you get the four possible solutions you can get.

The following screenshot shows the output of the application when both tasks throw an exception:




## There's more...

The `ThreadPoolExecutor` class provides another version of the `invokeAny()` method:

- `invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`: This method executes all the tasks and returns the result of the first one that finishes without throwing an exception, if it finishes before the given timeout passes. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

## See also

- The *Running multiple tasks and processing all the results* recipe in *Chapter 4*, *Thread Executors*

# Running multiple tasks and processing all the results

The Executor framework allows you to execute concurrent tasks without worrying about thread creation and execution. It provides you the `Future` class that you can use to control the status and get the results of any task executed in an executor.

When you want to wait for the finalization of a task, you can use the following two methods:

- The `isDone()` method of the `Future` interface returns `true` if the task has finished its execution.
- The `awaitTermination()` method of the `ThreadPoolExecutor` class puts the thread to sleep until all the tasks have finished their execution after a call to the `shutdown()` method.

These two methods have some drawbacks. With the first one, you can only control the completion of a task, and with the second one, you have to shutdown the executor to wait for a thread, otherwise the method's call returns immediately.

The `ThreadPoolExecutor` class provides a method that allows you to send to the executor a list of tasks and wait for the finalization of all the tasks in the list. In this recipe, you will learn how to use this feature by implementing an example with three tasks executed and their results printed out when they finish.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Result` to store the results generated in the concurrent tasks of this example.

```
public class Result {
```

2. Declare two private attributes. One `String` attribute named `name` and one `int` attribute named `value`.

```
  private String name;
  private int value;
```

3. Implement the corresponding `get()` and `set()` methods to set and return the value of the name and value attributes.

```
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getValue() {
```

```
    return value;
  }
  public void setValue(int value) {
    this.value = value;
  }
```

4. Create a class named `Task` that implements the `Callable` interface parameterized with the `Result` class.

```
public class Task implements Callable<Result> {
```

5. Declare a private `String` attribute named `name`.

```
  private String name;
```

6. Implement the constructor of the class that initializes its attribute.

```
  public Task(String name) {
    this.name=name;
  }
```

7. Implement the `call()` method of the class. In this case, this method will return a `Result` object.

```
  @Override
  public Result call() throws Exception {
```

8. First, write a message to the console to indicate that the task is starting.

```
    System.out.printf("%s: Staring\n",this.name);
```

9. Then, wait for a random period of time.

```
    try {
      long duration=(long)(Math.random()*10);
      System.out.printf("%s: Waiting %d seconds for results.\
n",this.name,duration);
      TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

10. To generate an `int` value to return in the `Result` object, calculate the sum of five random numbers.

```
    int value=0;
    for (int i=0; i<5; i++){
      value+=(int)(Math.random()*100);

    }
```

11. Create a `Result` object and initialize it with the name of this task and the result of the operation done earlier.

```
    Result result=new Result();
    result.setName(this.name);
    result.setValue(value);
```

12. Write a message to the console to indicate that the task has finished.

```
    System.out.println(this.name+": Ends");
```

13. Return the `Result` object.

```
    return result;
  }
```

14. Finally, implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

15. Create a `ThreadPoolExecutor` object using the `newCachedThreadPool()` method of the `Executors` class.

```
    ExecutorService executor=(ExecutorService)Executors.
newCachedThreadPool();
```

16. Create a list of `Task` objects. Create three `Task` objects and save them on that list.

```
    List<Task> taskList=new ArrayList<>();
    for (int i=0; i<3; i++){
      Task task=new Task(i);
      taskList.add(task);
    }
```

17. Create a list of `Future` objects. These objects are parameterized with the `Result` class.

```
    List<Future<Result>>resultList=null;
```

18. Call the `invokeAll()` method of the `ThreadPoolExecutor` class. This class will return the list of the `Future` objects created earlier.

```
    try {
      resultList=executor.invokeAll(taskList);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

19. Finalize the executor using the `shutdown()` method.

```
executor.shutdown();
```

20. Write the results of the tasks processing the list of the `Future` objects.

```
System.out.println("Main: Printing the results");
for (int i=0; i<resultList.size(); i++){
  Future<Result> future=resultList.get(i);
  try {
    Result result=future.get();
    System.out.println(result.getName()+": "+result.
getValue());
  } catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
  }
}
```

## How it works...

In this recipe, you have learned how to send a list of tasks to an executor and wait for the finalization of all of them using the `invokeAll()` method. This method receives a list of the `Callable` objects and returns a list of the `Future` objects. This list will have a `Future` object per task in the list. The first object in the list of the `Future` objects will be the object that controls the first task in the list of the `Callable` objects, and so on.

The first point to take into consideration is that the type of data used for the parameterization of the `Future` interface in the declaration of the list that stores the result objects must be compatible with the one used to parameterized the `Callable` objects. In this case, you have used the same type of data: the `Result` class.

Another important point about the `invokeAll()` method is that you will use the `Future` objects only to get the results of the tasks. As the method finishes when all the tasks have finished, if you call the `isDone()` method of the `Future` objects that is returned, all the calls will return the `true` value.

## There's more...

The `ExecutorService` class provides another version of the `invokeAll()` method:

- `invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`: This method executes all the tasks and returns the result of their execution when all of them have finished, if they finish before the given timeout passes. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

## See also

- The *Executing tasks in an executor that returns a result* recipe in *Chapter 4*, *Thread Executors*
- The *Running multiple tasks and processing the first result* recipe in *Chapter 4*, *Thread Executors*

# Running a task in an executor after a delay

The Executor framework provides the `ThreadPoolExecutor` class to execute `Callable` and `Runnable` tasks with a pool of threads, which avoid you all the thread creation operations. When you send a task to the executor, it's executed as soon as possible, according to the configuration of the executor. There are used cases when you are not interested in executing a task as soon as possible. You may want to execute a task after a period of time or to execute a task periodically. For these purposes, the Executor framework provides the `ScheduledThreadPoolExecutor` class.

In this recipe, you will learn how to create `ScheduledThreadPoolExecutor` and how to use it to schedule execution of a task after a given period of time.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` that implements the `Callable` interface parameterized with the `String` class.

   ```
   public class Task implements Callable<String> {
   ```

2. Declare a private `String` attribute named `name` that will store the name of the task.

   ```
     private String name;
   ```

3. Implement the constructor of the class that initializes the `name` attribute.

   ```
     public Task(String name) {
       this.name=name;
     }
   ```

4. Implement the `call()` method. Write a message to the console with the actual date and return a text, for example, `Hello, world`.

   ```
   public String call() throws Exception {
     System.out.printf("%s: Starting at : %s\n",name,new Date());
     return "Hello, world";
   }
   ```

5. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

   ```
   public class Main {
     public static void main(String[] args) {
   ```

6. Create an executor of the `ScheduledThreadPoolExecutor` class using the `newScheduledThreadPool()` method of the `Executors` class passing `1` as a parameter.

   ```
       ScheduledThreadPoolExecutor executor=(ScheduledThreadPoolExecu
   tor)Executors.newScheduledThreadPool(1);
   ```

7. Initialize and start a few tasks (five in our case) with the `schedule()` method of the `ScheduledThreadPoolExecutor` instance.

   ```
       System.out.printf("Main: Starting at: %s\n",new Date());
       for (int i=0; i<5; i++) {
         Task task=new Task("Task "+i);
         executor.schedule(task,i+1 , TimeUnit.SECONDS);
       }
   ```

8. Request the finalization of the executor using the `shutdown()` method.

   ```
       executor.shutdown();
   ```

9. Wait for the finalization of all the tasks using the `awaitTermination()` method of the executor.

   ```
       try {
         executor.awaitTermination(1, TimeUnit.DAYS);
       } catch (InterruptedException e) {
         e.printStackTrace();
       }
   ```

10. Write a message to indicate the time when the program finishes.

    ```
        System.out.printf("Main: Ends at: %s\n",new Date());
    ```

## How it works...

The key point of this example is the `Main` class and the management of `ScheduledThreadPoolExecutor`. As with class `ThreadPoolExecutor`, to create a scheduled executor, Java recommends the utilization of the `Executors` class. In this case, you have to use the `newScheduledThreadPool()` method. You have passed the number `1` as a parameter to this method. This parameter is the number of threads you want to have in the pool.

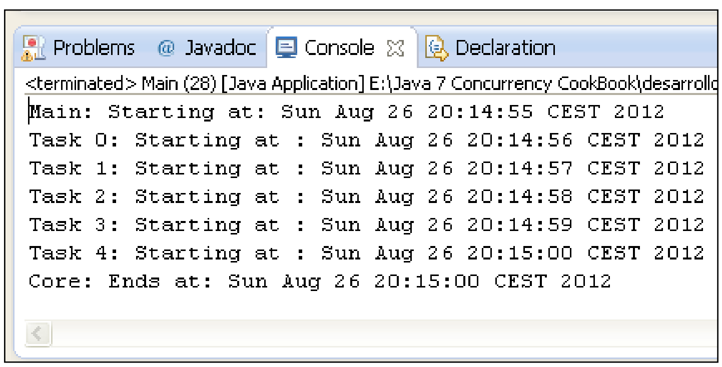
To execute a task in this scheduled executor after a period of time, you have to use the `schedule()` method. This method receives the following three parameters:

- The task you want to execute
- The period of time you want the task to wait before its execution
- The unit of the period of time, specified as a constant of the `TimeUnit` class

In this case, each task will wait for a number of seconds (`TimeUnit.SECONDS`) equal to its position in the array of tasks plus one.

> If you want to execute a task at a given time, calculate the difference between that date and the current date and use that difference as the delay of the task.

The following screenshot shows the output of an execution of this example:




You can see how the tasks start their execution one per second. All the tasks are sent to the executor at the same time, but each one with a delay of 1 second later than the previous task.

## There's more...

You can also use the `Runnable` interface to implement the tasks, because the `schedule()` method of the `ScheduledThreadPoolExecutor` class accepts both types of tasks.

Although the `ScheduledThreadPoolExecutor` class is a child class of the `ThreadPoolExecutor` class and, therefore, inherits all its features, Java recommends the utilization of `ScheduledThreadPoolExecutor` only for scheduled tasks.

Finally, you can configure the behavior of the `ScheduledThreadPoolExecutor` class when you call the `shutdown()` method and there are pending tasks waiting for the end of their delay time. The default behavior is that those tasks will be executed despite the finalization of the executor. You can change this behavior using the `setExecuteExistingDelayedTasksAfterShutdownPolicy()` method of the `ScheduledThreadPoolExecutor` class. With `false`, at the time of `shutdown()`, pending tasks won't get executed.

## See also

- The *Executing tasks in an executor that returns a result* recipe in *Chapter 4*, *Thread Executors*

# Running a task in an executor periodically

The Executor framework provides the `ThreadPoolExecutor` class to execute concurrent tasks using a pool of threads that avoids you all the thread creation operations. When you send a task to the executor, according to its configuration, it executes the task as soon as possible. When it ends, the task is deleted from the executor and, if you want to execute them again, you have to send it again to the executor.

But the Executor framework provides the possibility of executing periodic tasks through the `ScheduledThreadPoolExecutor` class. In this recipe, you will learn how to use this functionality of that class to schedule a periodic task.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable {
```

2. Declare a private `String` attribute named `name` that will store the name of the task.

```
  private String name;
```

3. Implement the constructor of the class that initializes that attribute.

```
  public Task(String name) {
    this.name=name;
  }
```

4. Implement the `run()` method. Write a message to the console with the actual date to verify that the task is executed within the specified period.

```
  @Override
  public String call() throws Exception {
    System.out.printf("%s: Starting at : %s\n",name,new Date());
    return "Hello, world";
  }
```

5. Implement the main class of the example by creating a class named `Main` and implement the `main()` method in it.

```
public class Main {
  public static void main(String[] args) {
```

6. Create `ScheduledThreadPoolExecutor` using the `newScheduledThreadPool()` method of the `Executors` class. Pass the number `1` as the parameter to that method.

```
    ScheduledExecutorService executor=Executors.
newScheduledThreadPool(1);
```

7. Write a message to the console with the actual date.

```
    System.out.printf("Main: Starting at: %s\n",new Date());
```

8. Create a new `Task` object.

```
    Task task=new Task("Task");
```

9. Send it to the executor using the `scheduledAtFixRate()` method. Use as parameters the task created earlier, the number one, the number two, and the constant `TimeUnit.SECONDS`. This method returns a `ScheduledFuture` object that you can use to control the status of the task.

```
    ScheduledFuture<?> result=executor.scheduleAtFixedRate(task,
1, 2, TimeUnit.SECONDS);
```

10. Create a loop with 10 steps to write the time remaining for the next execution of the task. In the loop, use the `getDelay()` method of the `ScheduledFuture` object to get the number of milliseconds until the next execution of the task.

```
    for (int i=0; i<10; i++){
      System.out.printf("Main: Delay: %d\n",result.
getDelay(TimeUnit.MILLISECONDS));
Sleep the thread during 500 milliseconds.
      try {
        TimeUnit.MILLISECONDS.sleep(500);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

11. Finish the executor using the `shutdown()` method.

```
    executor.shutdown();
```

12. Put the thread to sleep for 5 seconds to verify that the periodic tasks have finished.

```
    try {
      TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

13. Write a message to indicate the end of the program.

```
    System.out.printf("Main: Finished at: %s\n",new Date());
```

## How it works...

When you want to execute a periodic task using the Executor framework, you need a `ScheduledExecutorService` object. To create it (as with every executor), Java recommends the use of the `Executors` class. This class works as a factory of executor objects. In this case, you should use the `newScheduledThreadPool()` method to create a `ScheduledExecutorService` object. That method receives as a parameter the number of threads of the pool. As you have only one task in this example, you have passed the value `1` as a parameter.

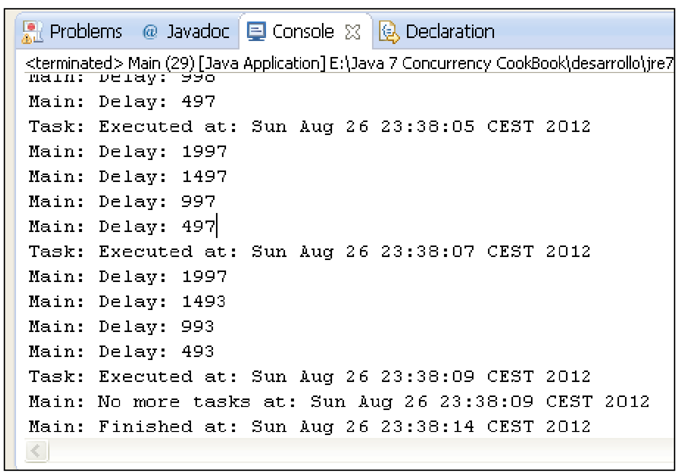
Once you have the executor needed to execute a periodic task, you send the task to the executor. You have used the `scheduledAtFixedRate()` method. This method accepts four parameters: the task you want to execute periodically, the delay of time until the first execution of the task, the period between two executions, and the time unit of the second and third parameters. It's a constant of the `TimeUnit` class. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

An important point to consider is that the period between two executions is the period of time between these two executions that begins. If you have a periodic task that takes 5 sceconds to execute and you put a period of 3 seconds, you will have two instances of the task executing at a time.

The method `scheduleAtFixedRate()` returns a `ScheduledFuture` object, which extends the `Future` interface, with methods to work with scheduled tasks. `ScheduledFuture` is a parameterized interface. In this example, as your task is a `Runnable` object that is not parameterized, you have to parameterize them with the `?` symbol as a parameter.

You have used one method of the `ScheduledFuture` interface. The `getDelay()` method returns the time until the next execution of the task. This method receives a `TimeUnit` constant with the time unit in which you want to receive the results.

The following screenshot shows the output of an execution of the example:




You can see the task executing every 2 seconds (denoted with `Task:` prefix) and the delay written in the console every 500 milliseconds. That's how long the main thread has been put to sleep. When you shut down the executor, the scheduled task ends its execution and you don't see more messages in the console.

## There's more...

`ScheduledThreadPoolExecutor` provides other methods to schedule periodic tasks. It is the `scheduleWithFixedRate()` method. It has the same parameters as the `scheduledAtFixedRate()` method, but there is a difference worth noticing. In the `scheduledAtFixedRate()` method, the third parameter determines the period of time between the starting of two executions. In the `scheduledWithFixedRate()` method, parameter determines the period of time between the end of an execution of the task and the beginning of the next execution.

You can also configure the behavior of an instance of the `ScheduledThreadPoolExecutor` class with the `shutdown()` method. The default behavior is that the scheduled tasks finish when you call that method. You can change this behavior using the `setContinueExistingPeriodicTasksAfterShutdownPolicy()` method of the `ScheduledThreadPoolExecutor` class with a `true` value. The periodic tasks won't finish upon calling the `shutdown()` method.

## See also

- The *Creating a thread executor* recipe in *Chapter 4*, *Thread Executors*
- The *Running a task in an executor after a delay* recipe in *Chapter 4*, *Thread Executors*

# Canceling a task in an executor

When you work with an executor, you don't have to manage threads. You only implement the `Runnable` or `Callable` tasks and send them to the executor. It's the executor that's responsible for creating threads, managing them in a thread pool, and finishing them if they are not needed. Sometimes, you may want to cancel a task that you sent to the executor. In that case, you can use the `cancel()` method of `Future` that allows you to make that cancellation operation. In this recipe, you will learn how to use this method to cancel the tasks that you have sent to an executor.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` and specify that it implements the `Callable` interface parameterized with the `String` class. Implement the `call()` method. Write a message to the console and put it to sleep for 100 milliseconds inside an infinite loop.

```
public class Task implements Callable<String> {
  @Override
  public String call() throws Exception {
    while (true){
      System.out.printf("Task: Test\n");
      Thread.sleep(100);
    }
  }
```

2. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

3. Create a `ThreadPoolExecutor` object using the `newCachedThreadPool()` method of the `Executors` class.

```
    ThreadPoolExecutor executor=(ThreadPoolExecutor)Executors.
newCachedThreadPool();
```

4. Create a new `Task` object.

```
    Task task=new Task();
```

5. Send the task to the executor using the `submit()` method.

```
    System.out.printf("Main: Executing the Task\n");
    Future<String> result=executor.submit(task);
```

6. Put the main task to sleep for 2 seconds.

```
    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

7. Cancel the execution of the task using the `cancel()` method of the `Future` object named `result` returned by the `submit()` method. Pass the `true` value as a parameter of the `cancel()` method.

```
    System.out.printf("Main: Canceling the Task\n");
    result.cancel(true);
```

8. Write to the console the result of a calling to the methods `isCancelled()` and `isDone()` to verify that the task has been canceled and hence, is already done.

```
System.out.printf("Main: Canceled: %s\n",result.isCanceled());
System.out.printf("Main: Done: %s\n",result.isDone());
```

9. Finish the executor with the `shutdown()` method and write a message indicating the finalization of the program.

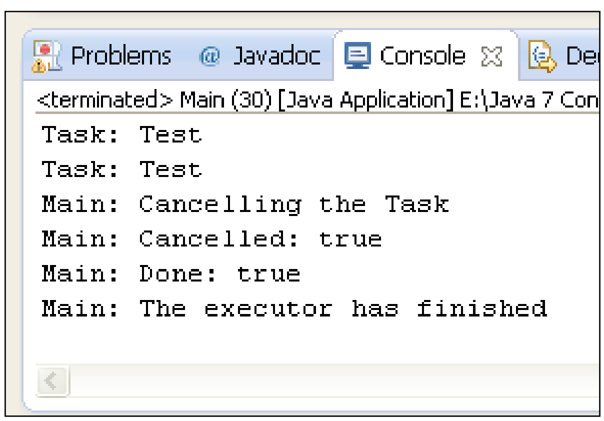
```
executor.shutdown();
System.out.printf("Main: The executor has finished\n");
```

## How it works...

You use the `cancel()` method of the `Future` interface when you want to cancel a task that you have sent to an executor. Depending on the parameter of the `cancel()` method and the status of the task, the behavior of this method is different:

- If the task has finished or has been canceled earlier or it can't be canceled for other reasons, the method will return the `false` value and the task won't be canceled.
- If the task is waiting in the executor to get a `Thread` object that will execute it, the task is canceled and never begins its execution. If the task is already running, it depends on the parameter of the method. The `cancel()` method receives a `Boolean` value as a parameter. If the value of that parameter is `true` and the task is running, it will be canceled. If the value of the parameter is `false` and the task is running, it won't be canceled.

The following screenshot shows the output of an execution of this example:



## There's more...

If you use the `get()` method of a `Future` object that controls a task that has been canceled, the `get()` method will throw a `CancellationException` exception.

## See also

- The *Executing tasks in an executor that returns a result* recipe in *Chapter 4, Thread Executors*

# Controlling a task finishing in an executor

The `FutureTask` class provides a method called `done()` that allows you to execute some code after the finalization of a task executed in an executor. It can be used to make some post-process operations, generating a report, sending results by e-mail, or releasing some resources. This method is called internally by the `FutureTask` class when the execution of the task that this `FutureTask` object is controlling finishes. The method is called after the result of the task is set and its status is changed to the `isDone` status, regardless of whether the task has been canceled or finished normally.

By default, this method is empty. You can override the `FutureTask` class and implement this method to change this behavior. In this recipe, you will learn how to override this method to execute code after the finalization of the tasks.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project..

## How to do it...

Follow these steps to implement the example:

1. Create a class named `ExecutableTask` and specify that it implements the `Callable` interface parameterized with the `String` class.

   ```
   public class ExecutableTask implements Callable<String> {
   ```

2. Declare a private `String` attribute named `name`. It will store the name of the task. Implement the method `getName()` to return the value of this attribute.

   ```
     private String name;
     public String getName(){
       return name;
     }
   ```

3. Implement the constructor of the class to initialize the name of the task.

```
  public ExecutableTask(String name){
    this.name=name;
  }
```

4. Implement the `call()` method. Put the task to sleep for a random period of time and return a message with the name of the task.

```
  @Override
  public String call() throws Exception {
    try {
      long duration=(long)(Math.random()*10);
      System.out.printf("%s: Waiting %d seconds for results.\
n",this.name,duration);
      TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
    }
    return "Hello, world. I'm "+name;
  }
```

5. Implement a class named `ResultTask` that extends the `FutureTask` class parameterized with the `String` class.

```
public class ResultTask extends FutureTask<String> {
```

6. Declare a private `String` attribute named `name`. It will store the name of the task.

```
  private String name;
```

7. Implement the constructor of the class. It has to receive a `Callable` object as a parameter. Call the constructor of the parent class and initialize the `name` attribute using the attribute of the task received.

```
  public ResultTask(Callable<String> callable) {
    super(callable);
    this.name=((ExecutableTask)callable).getName();
  }
```

8. Override the `done()` method. Check the value of the `isCancelled()` method and write a different message to the console depending on the returned value.

```
  @Override
  protected void done() {
    if (isCancelled()) {
      System.out.printf("%s: Has been canceled\n",name);
    } else {
      System.out.printf("%s: Has finished\n",name);
    }
  }
```

9. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

10. Create `ExecutorService` using the `newCachedThreadPool()` method of the `Executors` class.

```
    ExecutorService executor=(ExecutorService)Executors.
newCachedThreadPool();
```

11. Create an array to store five `ResultTask` objects.

```
    ResultTask resultTasks[]=new ResultTask[5];
```

12. Initialize the `ResultTask` objects. For each position in the array, first, you have to create `ExecutorTask` and then `ResultTask` using that object. Then, send to the executor `ResultTask` using the `submit()` method.

```
    for (int i=0; i<5; i++) {
      ExecutableTask executableTask=new ExecutableTask("Task "+i);
      resultTasks[i]=new ResultTask(executableTask);
      executor.submit(resultTasks[i]);
    }
```

13. Put the main thread to sleep for 5 seconds.

```
    try {
      TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e1) {
      e1.printStackTrace();
    }
```

14. Cancel all the tasks you have sent to the executor.

```
    for (int i=0; i<resultTasks.length; i++) {
      resultTasks[i].cancel(true);
    }
```

15. Write to the console the result of those tasks that haven't been canceled using the `get()` method of the `ResultTask` objects.

```
    for (int i=0; i<resultTasks.length; i++) {
      try {
        if (!resultTasks[i].isCanceled()){
          System.out.printf("%s\n",resultTasks[i].get());
        }
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }    }
```

16. Finish the executor using the `shutdown()` method.

```
    executor.shutdown();
  }
}
```

## How it works...

The `done()` method is called by the `FutureTask` class when the task that is being controlled finishes its execution. In this example, you have implemented a `Callable` object, the `ExecutableTask` class, and then, a subclass of the `FutureTask` class that controls the execution of the `ExecutableTask` objects.

The `done()` method is called internally by the `FutureTask` class after establishing the return value and changing the status of the task to the `isDone` status. You can't change the result value of the task or change its status, but you can close resources used by the task, write log messages, or send notifications.

## See also

- The *Executing tasks in an executor that returns a result* recipe in *Chapter 4, Thread Executors*

# Separating the launching of tasks and the processing of their results in an executor

Normally, when you execute concurrent tasks using an executor, you will send `Runnable` or `Callable` tasks to the executor and get `Future` objects to control the method. You can find situations, where you need to send the tasks to the executor in one object and process the results in another one. For such situations, the Java Concurrency API provides the `CompletionService` class.

This `CompletionService` class has a method to send the tasks to an executor and a method to get the `Future` object for the next task that has finished its execution. Internally, it uses an `Executor` object to execute the tasks. This behavior has the advantage to share a `CompletionService` object, and sends tasks to the executor so the others can process the results. The limitation is that the second object can only get the `Future` objects for those tasks that have finished its execution, so these `Future` objects can only be used to get the results of the tasks.

In this recipe, you will learn how to use the `CompletionService` class to separate launching tasks in an executor from processing their results.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `ReportGenerator` and specify that it implements the `Callable` interface parameterized with the `String` class.

    ```
    public class ReportGenerator implements Callable<String> {
    ```

2. Declare two private `String` attributes named `sender` and `title` that will represent data for the report.

    ```
      private String sender;
      private String title;
    ```

3. Implement the constructor of the class that initializes the two attributes.

    ```
      public ReportGenerator(String sender, String title){
        this.sender=sender;
        this.title=title;
      }
    ```

4. Implement the `call()` method. First, put the thread to sleep for a random period of time.

    ```
      @Override
      public String call() throws Exception {
        try {
          Long duration=(long)(Math.random()*10);
          System.out.printf("%s_%s: ReportGenerator: Generating a
    report during %d seconds\n",this.sender,this.title,duration);
          TimeUnit.SECONDS.sleep(duration);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
    ```

5. Then, generate the report as a string with the sender and title attributes and return that string.

    ```
        String ret=sender+": "+title;
        return ret;
      }
    ```

6. Create a class named `ReportRequest` and specify that it implements the `Runnable` interface. This class will simulate `some report` requests.

```
public class ReportRequest implements Runnable {
```

7. Declare a private `String` attribute named `name` to store the name of `ReportRequest`.

```
  private String name;
```

8. Declare a private `CompletionService` attribute named `service`. The `CompletionService` interface is a parameterized interface. Use the `String` class.

```
  private CompletionService<String> service;
```

9. Implement the constructor of the class that initializes the two attributes.

```
  public ReportRequest(String name, CompletionService<String>
service){
    this.name=name;
    this.service=service;
  }
```

10. Implement the `run()` method. Create three `ReportGenerator` objects and send them to the `CompletionService` object using the `submit()` method.

```
  @Override
  public void run() {

      ReportGenerator reportGenerator=new ReportGenerator(name,
"Report");
      service.submit(reportGenerator);

  }
```

11. Create a class named `ReportProcessor`. This class will get the results of the `ReportGenerator` tasks. Specify that it implements the `Runnable` interface.

```
public class ReportProcessor implements Runnable {
```

12. Declare a private `CompletionService` attribute named `service`. As the `CompletionService` interface is a parameterized interface, use the `String` class as parameter of this `CompletionService` interface.

```
  private CompletionService<String> service;
```

13. Declare a private `boolean` attribute named `end`.

```
  private boolean end;
```

14. Implement the constructor of the class to initialize the two attributes.

```
public ReportProcessor (CompletionService<String> service){
  this.service=service;
  end=false;
}
```

15. Implement the `run()` method. While the attribute `end` is `false`, call the `poll()` method of the `CompletionService` interface to get the `Future` object of the next task executed by the completion service that has finished.

```
  @Override
public void run() {
  while (!end){
    try {
      Future<String> result=service.poll(20, TimeUnit.SECONDS);
```

16. Then, get the results of the task using the `get()` method of the `Future` object and write those results to the console.

```
        if (result!=null) {
          String report=result.get();
          System.out.printf("ReportReceiver: Report Received: 
%s\n",report);
        }
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.printf("ReportSender: End\n");
  }
```

17. Implement the `setEnd()` method that modifies the value of the end attribute.

```
  public void setEnd(boolean end) {
    this.end = end;
  }
```

18. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

19. Create `ThreadPoolExecutor` using the `newCachedThreadPool()` method of the `Executors` class.

```
    ExecutorService executor=(ExecutorService)Executors.
newCachedThreadPool();
```

20. Create `CompletionService` using the executor created earlier as a parameter of the constructor.

```
    CompletionService<String> service=new ExecutorCompletionServic
e<>(executor);
```

21. Create two `ReportRequest` objects and the threads to execute them.

```
    ReportRequest faceRequest=new ReportRequest("Face", service);
    ReportRequest onlineRequest=new ReportRequest("Online",
service);
    Thread faceThread=new Thread(faceRequest);
    Thread onlineThread=new Thread(onlineRequest);
```

22. Create a `ReportProcessor` object and the thread to execute it.

```
    ReportProcessor processor=new ReportProcessor(service);
    Thread senderThread=new Thread(processor);
```

23. Start the three threads.

```
    System.out.printf("Main: Starting the Threads\n");
    faceThread.start();
    onlineThread.start();
    senderThread.start();
```

24. Wait for the finalization of the `ReportRequest` threads.

```
    try {
      System.out.printf("Main: Waiting for the report
generators.\n");
      faceThread.join();
      onlineThread.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

25. Finish the executor using the `shutdown()` method and wait for the finalization of the tasks with the `awaitTermination()` method.

```
    System.out.printf("Main: Shutting down the executor.\n");
    executor.shutdown();
    try {
      executor.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

26. Finish the execution of the `ReportSender` object setting the value of its end attribute to `true`.

```
processor.setEnd(true);
System.out.println("Main: Ends");
```

## How it works...

In the main class of the example, you have created `ThreadPoolExecutor` using the `newCachedThreadPool()` method of the `Executors` class. Then, you have used that object to initialize a `CompletionService` object because the completion service uses an executor to execute its tasks. To execute a task using the completion service, you use the `submit()` method as in the `ReportRequest` class.

When one of these tasks is executed when the completion service finishes its execution, the completion service stores the `Future` object used to control its execution in a queue. The `poll()` method accesses this queue to see if there is any task that has finished its execution and, if so, returns the first element of that queue which is a `Future` object for a task that has finished its execution. When the `poll()` method returns a `Future` object, it deletes it from the queue. In this case, you have passed two attributes to that method to indicate the time you want to wait for the finalization of a task, in case the queue with the results of the finished tasks is empty.

Once the `CompletionService` object is created, you create two `ReportRequest` objects that execute three `ReportGenerator` tasks, each one in `CompletionService`, and a `ReportSender` task that will process the results generated by the tasks sent by the two `ReportRequest` objects.

## There's more...

The `CompletionService` class can execute `Callable` or `Runnable` tasks. In this example, you have used `Callable`, but you could also send `Runnable` objects. Since `Runnable` objects don't produce a result, the philosophy of the `CompletionService` class doesn't apply in such cases.

This class also provides two other methods to obtain the `Future` objects of the finished tasks. These methods are as follows:

- `poll()`: The version of the `poll()` method without arguments checks if there are any `Future` objects in the queue. If the queue is empty, it returns `null` immediately. Otherwise, it returns its first element and removes it from the queue.
- `take()`: This method, without arguments, checks if there are any `Future` objects in the queue. If it is empty, it blocks the thread until the queue has an element. When the queue has elements, it returns and deletes its first element from the queue.

## See also

- The *Executing tasks in an executor that returns a result* recipe in *Chapter 4, Thread Executor*

# Controlling rejected tasks of an executor

When you want to finish the execution of an executor, you use the `shutdown()` method to indicate that it should finish. The executor waits for the completion of the tasks that are running or waiting for their execution, and then finishes its execution.

If you send a task to an executor between the `shutdown()` method and the end of its execution, the task is rejected, because the executor no longer accepts new tasks. The `ThreadPoolExecutor` class provides a mechanism, which is called when a task is rejected.

In this recipe, you will learn how to manage rejecting tasks in an executor that is implementing with `RejectedExecutionHandler`.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `RejectedTaskController` that implements the `RejectedExecutionHandler` interface. Implement the `rejectedExecution()` method of that interface. Write to the console the name of the task that has been rejected and the name and status of the executor.

```
public class RejectedTaskController implements
RejectedExecutionHandler {
  @Override
  public void rejectedExecution(Runnable r, ThreadPoolExecutor
executor) {
    System.out.printf("RejectedTaskController: The task %s has
been rejected\n",r.toString());
    System.out.printf("RejectedTaskController: %s\n",executor.
toString());
    System.out.printf("RejectedTaskController: Terminating:
%s\n",executor.isTerminating());
    System.out.printf("RejectedTaksController: Terminated:
%s\n",executor.isTerminated());
  }
```

2. Implement a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable{
```

3. Declare a private `String` attribute named `name`. It will store the name of the task.

```
  private String name;
```

4. Implement the constructor of the class. It will initialize the attribute of the class.

```
  public Task(String name){
    this.name=name;
  }
```

5. Implement the `run()` method. Write a message to the console to indicate the starting of the method.

```
   @Override
  public void run() {
    System.out.println("Task "+name+": Starting");
```

6. Wait for a random period of time.

```
    try {
      long duration=(long)(Math.random()*10);
      System.out.printf("Task %s: ReportGenerator: Generating a 
report during %d seconds\n",name,duration);
      TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

7. Write a message to the console to indicate the finalization of the method.

```
    System.out.printf("Task %s: Ending\n",name);
  }
```

8. Override the `toString()` method. Return the name of the task.

```
  public String toString() {
    return name;
  }
```

9. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

10. Create a `RejectedTaskController` object to manage the rejected tasks.

```
    RejectecTaskController controller=new
RejectecTaskController();
```

11. Create `ThreadPoolExecutor` using the `newCachedThreadPool()` method of the `Executors` class.

```
    ThreadPoolExecutor executor=(ThreadPoolExecutor) Executors.
newCachedThreadPool();
```

12. Establish the rejected task controller of the executor.

```
    executor.setRejectedExecutionHandler(controller);
```

13. Create three tasks and send them to the executor.

```
    System.out.printf("Main: Starting.\n");
    for (int i=0; i<3; i++) {
      Task task=new Task("Task"+i);
      executor.submit(task);
    }
```

14. Shutdown the executor using the `shutdown()` method.

```
    System.out.printf("Main: Shutting down the Executor.\n");
    executor.shutdown();
```

15. Create another task and send it to the executor.

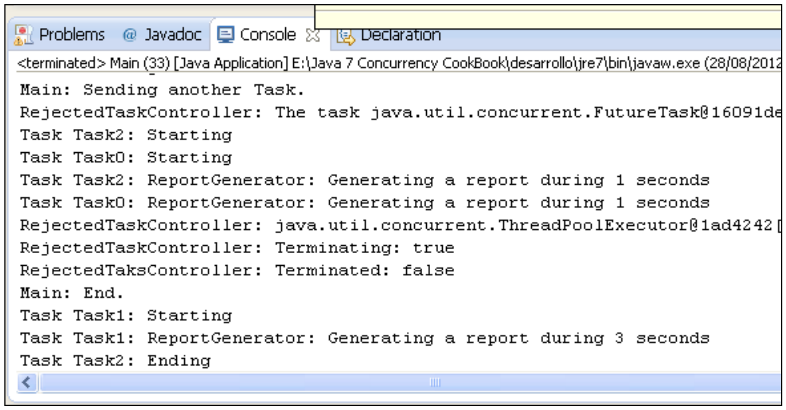
```
    System.out.printf("Main: Sending another Task.\n");
    Task task=new Task("RejectedTask");
    executor.submit(task);
```

16. Write a message to the console to indicate the finalization of the program.

```
    System.out.println("Main: End");
    System.out.printf("Main: End.\n");
```

## How it works...

In the following screenshot, you can see the result of an execution of the example:




You can see that the task is rejected when execution has been shut down and `RejectecTaskController` writes to the console information about the task and the executor.

To manage rejected tasks for an executor, you should create a class that implements the `RejectedExecutionHandler` interface. This interface has a method called `rejectedExecution()` with two parameters:

- A `Runnable` object that stores the task that has been rejected
- An `Executor` object that stores the executor that rejected the task

This method is called for every task that is rejected by the executor. You need to establish the handler of the rejected tasks using the `setRejectedExecutionHandler()` method of the `Executor` class.

## There's more...

When an executor receives a task to execute, it checks if the `shutdown()` method has been called. If so, it rejects the task. First, it looks for the handler established with `setRejectedExecutionHandler()`. If there's one, it calls the `rejectedExecution()` method of that class, otherwise it throws `RejectedExecutionExeption`. This is a runtime exception, so you don't need to put a `catch` clause to control it.

## See also

- The *Creating a thread executor* recipe in *Chapter 4*, *Thread Executors*

# 5
# Fork/Join Framework

In this chapter, we will cover:

- Creating a Fork/Join pool
- Joining the results of the tasks
- Running tasks asynchronously
- Throwing exceptions in the tasks
- Canceling a task

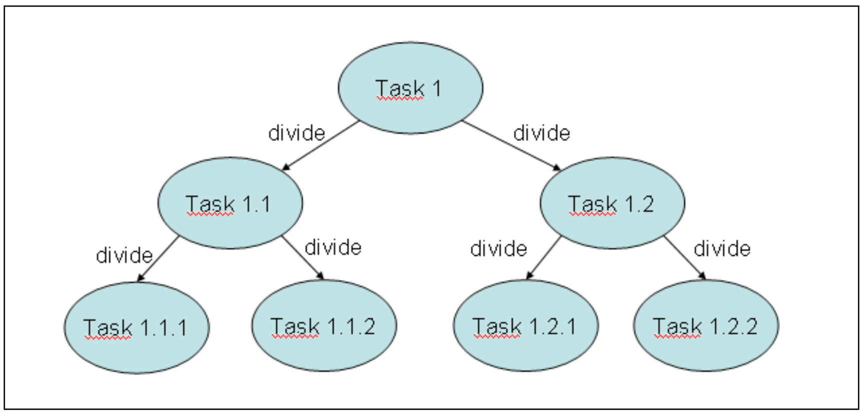
## Introduction

Normally, when you implement a simple, concurrent Java application, you implement some `Runnable` objects and then the corresponding `Thread` objects. You control the creation, execution, and status of those threads in your program. Java 5 introduced an improvement with the `Executor` and `ExecutorService` interfaces and the classes that implement them (for example, the `ThreadPoolExecutor` class).

The Executor framework separates the task creation and its execution. With it, you only have to implement the `Runnable` objects and use an `Executor` object. You send the `Runnable` tasks to the executor and it creates, manages, and finalizes the necessary threads to execute those tasks.

Java 7 goes a step further and includes an additional implementation of the `ExecutorService` interface oriented to a specific kind of problem. It's the **Fork/Join framework**.

This framework is designed to solve problems that can be broken into smaller tasks using the divide and conquer technique. Inside a task, you check the size of the problem you want to resolve and, if it's bigger than an established size, you divide it in smaller tasks that are executed using the framework. If the size of the problem is smaller than the established size, you solve the problem directly in the task and then, optionally, it returns a result. The following diagram summarizes this concept:




There is no formula to determine the reference size of a problem that determines if a task is subdivided or not, depending on its characteristics. You can use the number of elements to process in the task and an estimation of the execution time to determine the reference size. Test different reference sizes to choose the best one to your problem. You can consider `ForkJoinPool` as a special kind of `Executor`.

The framework is based on the following two operations:

- The **fork** operation: When you divide a task into smaller tasks and execute them using the framework
- The **join** operation: When a task waits for the finalization of the tasks it has created

The main difference between the Fork/Join and the Executor frameworks is the **work-stealing** algorithm. Unlike the Executor framework, when a task is waiting for the finalization of the subtasks it has created using the join operation, the thread that is executing that task (called **worker thread**) looks for other tasks that have not been executed yet and begins its execution. By this way, the threads take full advantage of their running time, thereby improving the performance of the application.

To achieve this goal, the tasks executed by the Fork/Join framework have the following limitations:

- Tasks can only use the `fork()` and `join()` operations as synchronization mechanisms. If they use other synchronization mechanisms, the worker threads can't execute other tasks when they are in the synchronization operation. For example, if you put a task to sleep in the Fork/Join framework, the worker thread that is executing that task won't execute another one during the sleeping time.

- Tasks should not perform I/O operations, such as read or write data in a file.
- Tasks can't throw checked exceptions. It has to include the code necessary to process them.

The core of the Fork/Join framework is formed by the following two classes:

- `ForkJoinPool`: It implements the `ExecutorService` interface and the work-stealing algorithm. It manages the worker threads and offers information about the status of the tasks and their execution.
- `ForkJoinTask`: It's the base class of the tasks that will execute in `ForkJoinPool`. It provides the mechanisms to execute the `fork()` and `join()` operations inside a task and the methods to control the status of the tasks. Usually, to implement your Fork/Join tasks, you will implement a subclass of two subclasses of this class: `RecursiveAction` for tasks with no return result and `RecursiveTask` for tasks that return one.

This chapter presents five recipes that show you how to work efficiently with the Fork/Join framework.

# Creating a Fork/Join pool

In this recipe, you will learn how to use the basic elements of the Fork/Join framework. This includes:

- Creating a `ForkJoinPool` object to execute the tasks
- Creating a subclass of `ForkJoinTask` to be executed in the pool

The main characteristics of the Fork/Join framework you're going to use in this example are as follows:

- You will create `ForkJoinPool` using the default constructor.
- Inside the task, you will use the structure recommended by the Java API documentation:

```
If (problem size > default size){
  tasks=divide(task);
  execute(tasks);
} else {
  resolve problem using another algorithm;
}
```

- You will execute the tasks in a synchronized way. When a task executes two or more subtasks, it waits for their finalizations. By this way, the thread that was executing that task (called worker-thread) will look for other tasks to execute, taking full advantage of their execution time.
- The tasks you're going to implement won't return any result, so you'll take the `RecursiveAction` class as the base class for their implementation.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

In this recipe, you are going to implement a task to update the price of a list of products. The initial task will be responsible for updating all the elements in a list. You will use a size 10 as the reference size so, if a task has to update more than 10 elements, it divides the part of the list assigned to it in two parts and creates two tasks to update the prices of the products in respective parts.

Follow these steps to implement the example:

1. Create a class named `Product` that will store the name and price of a product.

```
public class Product {
```

2. Declare a private `String` attribute named `name` and a private `double` attribute named `price`.

```
  private String name;
  private double price;
```

3. Implement both the methods and establish the values of both attributes.

```
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public double getPrice() {
    return price;
  }

  public void setPrice(double price) {
```

```
    this.price = price;
  }
```

4. Create a class named `ProductListGenerator` to generate a list of random products.

```
public class ProductListGenerator {
```

5. Implement the `generate()` method. It receives an `int` parameter with the size of the list and returns a `List<Product>` object with the list of generated products.

```
  public List<Product> generate (int size) {
```

6. Create the object to return the list of products.

```
    List<Product> ret=new ArrayList<Product>();
```

7. Generate the list of products. Assign the same price to all of the products, for example, 10 to check that the program works well.

```
    for (int i=0; i<size; i++){
      Product product=new Product();
      product.setName("Product "+i);
      product.setPrice(10);
      ret.add(product);
    }
    return ret;
  }
```

8. Create a class named `Task`. Specify that it extends the `RecursiveAction` class.

```
public class Task extends RecursiveAction {
```

9. Declare the serial version UID of the class. This element is necessary, because the parent class of the `RecursiveAction` class, the `ForkJoinTask` class, implements the `Serializable` interface.

```
  private static final long serialVersionUID = 1L;
```

10. Declare a private `List<Product>` attribute named `products`.

```
  private List<Product> products;
```

11. Declare two private `int` attributes, named `first` and `last`. These attributes will determine the block of products this task has to process.

```
  private int first;
  private int last;
```

12. Declare a private `double` attribute named `increment` to store the increment of the price of the products.

```
  private double increment;
```

13. Implement the constructor of the class that will initialize all the attributes of the class.

```
  public Task (List<Product> products, int first, int last, double 
increment) {
    this.products=products;
    this.first=first;
    this.last=last;
    this.increment=increment;
  }
```

14. Implement the `compute()` method that will implement the logic of the task.

```
  @Override
  protected void compute() {
```

15. If the difference of the `last` and `first` attributes is smaller than 10 (the task has to update the price of less than 10 products), increment the price of that set or products using the `updatePrices()` method.

```
    if (last-first<10) {
      updatePrices();
```

16. If the difference between the `last` and `first` attributes is greater than or equal to 10, create two new `Task` objects, one to process the first half of products and the other to process the second half and execute them in `ForkJoinPool` using the `invokeAll()` method.

```
    } else {
      int middle=(last+first)/2;
      System.out.printf("Task: Pending tasks: 
%s\n",getQueuedTaskCount());
      Task t1=new Task(products, first,middle+1, increment);
      Task t2=new Task(products, middle+1,last, increment);
      invokeAll(t1, t2);
    }
```

17. Implement the `updatePrices()` method. This method updates the products that occupy the positions between the values of `first` and `last` attributes in the list of products.

```
  private void updatePrices() {
    for (int i=first; i<last; i++){
      Product product=products.get(i);
      product.setPrice(product.getPrice()*(1+increment));
    }
  }
```

18. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

19. Create a list of 10,000 products using the `ProductListGenerator` class.

```
    ProductListGenerator generator=new ProductListGenerator();
    List<Product> products=generator.generate(10000);
```

20. Create a new `Task` object to update the products of all the products of the list. The parameter `first` takes the value `0` and the `last` parameter takes the value `10,000` (the size of the products list).

```
      Task task=new Task(products, 0, products.size(), 0.20);
```

21. Create a `ForkJoinPool` object using the constructor without parameters.

```
    ForkJoinPool pool=new ForkJoinPool();
```

22. Execute the task in the pool using the `execute()` method.

```
    pool.execute(task);
```

23. Implement a block of code that shows information about the evolution of the pool every five milliseconds writing to the console the value of some parameters of the pool until the task finishes its execution.

```
    do {
      System.out.printf("Main: Thread Count: %d\n",pool.
getActiveThreadCount());
      System.out.printf("Main: Thread Steal: %d\n",pool.
getStealCount());
      System.out.printf("Main: Parallelism: %d\n",pool.
getParallelism());
      try {
        TimeUnit.MILLISECONDS.sleep(5);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    } while (!task.isDone());
```

24. Shut down the pool using the `shutdown()` method.

```
    pool.shutdown();
```

25. Check if the task has finished without errors with the `isCompletedNormally()` method and, in that case, write a message to the console.

```
    if (task.isCompletedNormally()){
      System.out.printf("Main: The process has completed 
normally.\n");
    }
```

26. The expected price of all products, after the increment, is 12. Write the name and the price of all the products that have a price difference of 12 to check that all of them have increased their price correctly.

```
    for (int i=0; i<products.size(); i++){
      Product product=products.get(i);
      if (product.getPrice()!=12) {
        System.out.printf("Product %s: %f\n",product.
getName(),product.getPrice());
      }
    }
```

27. Write a message to indicate the finalization of the program.

```
    System.out.println("Main: End of the program.\n");
```

## How it works...

In this example, you have created a `ForkJoinPool` object and a subclass of the `ForkJoinTask` class that you execute in the pool. To create the `ForkJoinPool` object, you have used the constructor without arguments, so it will be executed with its default configuration. It creates a pool with a number of threads equal to the number of processors of the computer. When the `ForkJoinPool` object is created, those threads are created and they wait in the pool until some tasks arrive for their execution.

Since the `Task` class doesn't return a result, it extends the `RecursiveAction` class. In the recipe, you have used the recommended structure for the implementation of the task. If the task has to update more than 10 products, it divides those set of elements into two blocks, creates two tasks, and assigns a block to each task. You have used the `first` and `last` attributes in the `Task` class to know the range of positions that this task has to update in the list of products. You have used the `first` and `last` attributes to use only one copy of the products list and not create different lists for each task.

To execute the subtasks that a task creates, it calls the `invokeAll()` method. This is a synchronous call, and the task waits for the finalization of the subtasks before continuing (potentially finishing) its execution. While the task is waiting for its subtasks, the worker thread that was executing it takes another task that was waiting for execution and executes it. With this behavior, the Fork/Join framework offers a more efficient task management than the `Runnable` and `Callable` objects themselves.

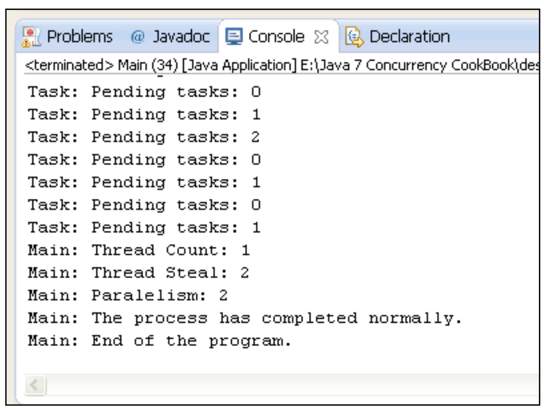
The `invokeAll()` method of the `ForkJoinTask` class is one of the main differences between the Executor and the Fork/Join framework. In the Executor framework, all the tasks have to be sent to the executor, while in this case, the tasks include methods to execute and control the tasks inside the pool. You have used the `invokeAll()` method in the `Task` class, that extends the `RecursiveAction` class that extends the `ForkJoinTask` class.

You have sent a unique task to the pool to update all the list of products using the `execute()` method. In this case, it's an asynchronous call, and the main thread continues its execution.

You have used some methods of the `ForkJoinPool` class to check the status and the evolution of the tasks that are running. The class includes more methods that can be useful for this purpose. See the *Monitoring a Fork/Join pool* recipe for a complete list of those methods.

Finally, like with the Executor framework, you should finish `ForkJoinPool` using the `shutdown()` method.

The following screenshot shows part of an execution of this example:




You can see the tasks finishing their work and the price of the products updates.

## There's more...

The `ForkJoinPool` class provides other methods to execute a task in. These methods are as follows:

- `execute (Runnable task)`: This is another version of the `execute()` method used in the example. In this case, you send a `Runnable` task to the `ForkJoinPool` class. Note that the `ForkJoinPool` class doesn't use the work-stealing algorithm with `Runnable` objects. It's only used with `ForkJoinTask` objects.

- `invoke(ForkJoinTask<T> task)`: While the `execute()` method makes an asynchronous call to the `ForkJoinPool` class, as you learned in the example, the `invoke()` method makes a synchronous call to the `ForkJoinPool` class. This call doesn't return until the task passed as a parameter finishes its execution.
- You also can use the `invokeAll()` and `invokeAny()` methods declared in the `ExecutorService` interface. These methods receive `Callable` objects as parameters. The `ForkJoinPool` class doesn't use the work-stealing algorithm with the `Callable` objects, so you'd be better off executing them using an executor.

The `ForkJoinTask` class also includes other versions of the `invokeAll()` method used in the example. These versions are as follows:

- `invokeAll(ForkJoinTask<?>... tasks)`: This version of the method uses a variable list of arguments. You can pass to it as parameters as many `ForkJoinTask` objects as you want.
- `invokeAll(Collection<T> tasks)`: This version of the method accepts a collection (for example, an `ArrayList` object, a `LinkedList` object, or a `TreeSet` object) of objects of a generic type `T`. This generic type `T` must be the `ForkJoinTask` class or a subclass of it.

Although the `ForkJoinPool` class is designed to execute an object of `ForkJoinTask`, you can also execute `Runnable` and `Callable` objects directly. You may also use the `adapt()` method of the `ForkJoinTask` class that accepts a `Callable` object or a `Runnable` object and returns a `ForkJoinTask` object to execute that task.

## See also

- The *Monitoring a Fork/Join pool* recipe in *Chapter 8*, *Testing concurrent applications*

# Joining the results of the tasks

The Fork/Join framework provides the ability of executing tasks that return a result. These kinds of tasks are implemented by the `RecursiveTask` class. This class extends the `ForkJoinTask` class and implements the `Future` interface provided by the Executor framework.

Inside the task, you have to use the structure recommended by the Java API documentation:

```
If (problem size > size){
  tasks=Divide(task);
  execute(tasks);
  groupResults()
  return result;
} else {
```

```
    resolve problem;
    return result;
  }
```

If the task has to resolve a problem bigger than a predefined size, you divide the problem in more subtasks and execute those subtasks using the Fork/Join framework. When they finish their execution, the initiating task obtains the results generated by all the subtasks, groups them, and returns the final result. Ultimately, when the initiating task executed in the pool finishes its execution, you obtain its result that is effectively the final result of the entire problem.

In this recipe, you will learn how to use this kind of problem solving with Fork/Join framework developing an application that looks for a word in a document. You will implement the following two kinds of tasks:

- A document task, which is going to search a word in a set of lines of a document
- A line task, which is going to search a word in a part of the document

All the tasks are going to return the number of appearances of the word in the part of the document or line they process.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Document`. It will generate a string matrix that will simulate a document.

   ```
   public class Document {
   ```

2. Create an array of strings with some words. This array will be used in the generation of the strings matrix.

   ```
   private String words[]={"the","hello","goodbye","packt", "java","t
   hread","pool","random","class","main"};
   ```

3. Implement the `generateDocument()` method. It receives as parameters the number of lines, the number of words per line, and the word the example is going to look for. It returns a matrix of strings.

   ```
     public String[][] generateDocument(int numLines, int numWords,
   String word){
   ```

4. First, create the necessary objects to generate the document: the `String` matrix and a `Random` object to generate random numbers.

   ```
       int counter=0;
       String document[][]=new String[numLines][numWords];
       Random random=new Random();
   ```

5. Fill the array with strings. Store in each position the string which is at a random position in the array of words and count the number of appearances of the word the program will look for in the generated array. You can use this value to check whether the program does its job properly.

```
    for (int i=0; i<numLines; i++){
      for (int j=0; j<numWords; j++) {
        int index=random.nextInt(words.length);
        document[i][j]=words[index];
        if (document[i][j].equals(word)){
          counter++;
        }
      }
    }
```

6. Write a message with the number of appearances of the word and return the matrix generated.

```
    System.out.println("DocumentMock: The word appears "+
counter+" times in the document");
    return document;
```

7. Create a class named `DocumentTask` and specify that it extends the `RecursiveTask` class parameterized with the `Integer` class. This class will implement the task that will calculate the number of appearances of the word in a set of lines.

```
public class DocumentTask extends RecursiveTask<Integer> {
```

8. Declare a private `String` matrix named `document` and two private `int` attributes named `start` and `end`. Declare also a private `String` attribute named `word`.

```
  private String document[][];
  private int start, end;
  private String word;
```

9. Implement the constructor of the class to initialize all its attributes.

```
  public DocumentTask (String document[][], int start, int end,
String word){
    this.document=document;
    this.start=start;
    this.end=end;
    this.word=word;
  }
```

10. Implement the `compute()` method. If the difference between the `end` and `start` attributes is smaller than 10, the task calculates the number of appearances of a word in the lines between those positions calling the `processLines()` method.

```
  @Override
  protected Integer compute() {
      int result;
    if (end-start<10){
      result=processLines(document, start, end, word);
```

11. Otherwise, divide the group of lines in two objects, create two new `DocumentTask` objects to process those two groups, and execute them in the pool using the `invokeAll()` method.

```
    } else {
      int mid=(start+end)/2;
      DocumentTask task1=new DocumentTask(document,start,mid,wo
rd);
      DocumentTask task2=new DocumentTask(document,mid,end,word);
      invokeAll(task1,task2);
```

12. Then, add the values returned by both tasks using the `groupResults()` method. Finally, return the result calculated by the task.

```
      try {
        result=groupResults(task1.get(),task2.get());
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
    return result;
```

13. Implement the `processLines()` method. It receives the string matrix, the `start` attribute, the `end` attribute, and the `word` attribute the task is searching for as parameters.

```
  private Integer processLines(String[][] document, int start, int
end,String word) {
```

14. For each line the task has to process, create a `LineTask` object to process the complete line, and store them in a list of tasks.

```
    List<LineTask> tasks=new ArrayList<LineTask>();
    for (int i=start; i<end; i++){
      LineTask task=new LineTask(document[i], 0, document[i].
length, word);
      tasks.add(task);
    }
```

15. Execute all the tasks in that list using the `invokeAll()` method.

```
invokeAll(tasks);
```

16. Sum the value returned by all these tasks and return the result.

```
int result=0;
for (int i=0; i<tasks.size(); i++) {
  LineTask task=tasks.get(i);
  try {
    result=result+task.get();
  } catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
  }
}
return new Integer(result);
```

17. Implement the `groupResults()` method. It adds two numbers and returns the result.

```
private Integer groupResults(Integer number1, Integer number2) {
  Integer result;
  result=number1+number2;
  return result;
}
```

18. Create a class named `LineTask` and specify that it extends the `RecursiveTask` class parameterized with the `Integer` class. This class will implement the task that will calculate the number of appearances of the word in a line.

```
public class LineTask extends RecursiveTask<Integer>{
```

19. Declare the serial version UID of the class. This element is necessary because the parent class of the `RecursiveTask` class, the `ForkJoinTask` class, implements the `Serializable` interface. Declare a private `String` array attribute named `line` and two private `int` attributes named `start` and `end`. Finally, declare a private `String` attribute named `word`.

```
  private static final long serialVersionUID = 1L;
  private String line[];
  private int start, end;
  private String word;
```

20. Implement the constructor of the class to initialize all its attributes.

```
  public LineTask(String line[], int start, int end, String word)
{
    this.line=line;
    this.start=start;
    this.end=end;
```

```
    this.word=word;
  }
```

21. Implement the `compute()` method of the class. If the difference between the `end` and `start` attributes is smaller than 100, the task searches for the word in the fragment of the line determined by the `start` and `end` attributes using the `count()` method.

```
  @Override
  protected Integer compute() {
    Integer result=null;
    if (end-start<100) {
      result=count(line, start, end, word);
```

22. Otherwise, divide the group of words in the line in two, create two new `LineTask` objects to process those two groups, and execute them in the pool using the `invokeAll()` method.

```
    } else {
      int mid=(start+end)/2;
      LineTask task1=new LineTask(line, start, mid, word);
      LineTask task2=new LineTask(line, mid, end, word);
      invokeAll(task1, task2);
```

23. Then, add the values returned by both tasks using the `groupResults()` method. Finally, return the result calculated by the task.

```
      try {
        result=groupResults(task1.get(),task2.get());
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
    return result;
```

24. Implement the `count()` method. It receives the string array with the complete line, the `star` attribute, the `end` attribute, and the `word` attribute the task is searching for as parameters.

```
  private Integer count(String[] line, int start, int end, String
word) {
```

25. Compare the words stored in the positions between the `start` and `end` attributes with the `word` attribute the task is searching for and if they are equal, increment a `counter` variable.

```
    int counter;
    counter=0;
    for (int i=start; i<end; i++){
      if (line[i].equals(word)){
```

```
            counter++;
          }
        }
```

26. To slow the execution of the example, put the task to sleep for 10 milliseconds.

```
        try {
          Thread.sleep(10);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
```

27. Return the value of the counter variable.

```
        return counter;
```

28. Implement the groupResults() method. It sums two numbers and returns the result.

```
      private Integer groupResults(Integer number1, Integer number2) {
        Integer result;
        result=number1+number2;
        return result;
      }
```

29. Implement the main class of the example by creating a class named Main with a main() method.

```
    public class Main{
      public static void main(String[] args) {
```

30. Create Document with 100 lines and 1,000 words per line using the DocumentMock class.

```
        DocumentMock mock=new DocumentMock();
        String[][] document=mock.generateDocument(100, 1000, "the");
```

31. Create a new DocumentTask object to update the products of the entire document. The parameter start takes the value 0 and the end parameter takes the value 100.

```
        DocumentTask task=new DocumentTask(document, 0, 100, "the");
```

32. Create a ForkJoinPool object using the constructor without parameters and execute the task in the pool using the execute() method.

```
        ForkJoinPool pool=new ForkJoinPool();
        pool.execute(task);
```

33. Implement a block of code that shows information about the progress of the pool writing every second to the console the value of some parameters of the pool until the task finishes its execution.

```
    do {
      System.out.printf("*****************************************
*\n");
      System.out.printf("Main: Parallelism: %d\n",pool.
getParallelism());
      System.out.printf("Main: Active Threads: %d\n",pool.
getActiveThreadCount());
      System.out.printf("Main: Task Count: %d\n",pool.
getQueuedTaskCount());
      System.out.printf("Main: Steal Count: %d\n",pool.
getStealCount());
      System.out.printf("*****************************************
*\n");
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    } while (!task.isDone());
```

34. Shut down the pool using the `shutdown()` method.

```
    pool.shutdown();
```

35. Wait for the finalization of the tasks using the `awaitTermination()` method.

```
    try {
      pool.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

36. Write the number of the appearances of the word in the document. Check that this number is the same as the number written by the `DocumentMock` class.

```
    try {
      System.out.printf("Main: The word appears %d in the
document",task.get());
    } catch (InterruptedException | ExecutionException e) {
      e.printStackTrace();
    }
```

## How it works...

In this example, you have implemented two different tasks:

- The `DocumentTask` class: A task of this class has to process a set of lines of the document determined by the `start` and `end` attributes. If this set of lines has a size smaller that 10, it creates `LineTask` per line, and when they finish their execution, it sums the results of those tasks and returns the result of the sum. If the set of lines the task has to process has a size of 10 or bigger, it divides the set in two and creates two `DocumentTask` objects to process those new sets. When those tasks finish their execution, the tasks sum their results and return that sum as a result.
- The `LineTask` class: A task of this class has to process a set of words of a line of the document. If this set of words is smaller than 100, the task searches the word directly in that set of words and returns the number of appearances of the word. Otherwise, it divides the set of words in two and creates two `LineTask` objects to process those sets. When those tasks finish their execution, the task sums the results of both tasks and returns that sum as a result.

In the `Main` class, you have created a `ForkJoinPool` object using the default constructor and you have executed in it a `DocumentTask` class that has to process a document of 100 lines and 1,000 words per line. This task is going to divide the problem using other `DocumentTask` objects and `LineTask` objects, and when all the tasks finish their execution, you can use the original task to get the total number of appearances of the word in the whole document. Since the tasks return a result, they extend the `RecursiveTask` class.

To obtain the result returned by `Task`, you have used the `get()` method. This method is declared in the `Future` interface implemented by the `RecursiveTask` class.

When you execute the program, you can compare the first and the last lines written in the console. The first line is the number of appearances of the word calculated when the document is generated and the last is the same number calculated by the Fork/Join tasks.

## There's more...

The `ForkJoinTask` class provides another method to finish execution of a task and returns a result, that is, the `complete()` method. This method accepts an object of the type used in the parameterization of the `RecursiveTask` class and returns that object as a result of the task when the `join()` method is called. It's use is recommended to provide results for asynchronous tasks.

Since the `RecursiveTask` class implements the `Future` interface, there's another version of the `get()` method:

- `get(long timeout, TimeUnit unit)`: This version of the `get()` method, if the result of the task isn't available, waits the specified time for it. If the specified period of time passes and the result isn't yet available, the method returns a `null` value. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

## See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*
- The *Monitoring a Fork/Join Pool* recipe in *Chapter 8*, *Testing concurrent applications*

# Running tasks asynchronously

When you execute `ForkJoinTask` in `ForkJoinPool`, you can do it in a synchronous or asynchronous way. When you do it in a synchronous way, the method that sends the task to the pool doesn't return until the task sent finishes its execution. When you do it in an asynchronous way, the method that sends the task to the executor returns immediately, so the task can continue with its execution.

You should be aware of a big difference between the two methods. When you use the synchronized methods, the task that calls one of these methods (for example, the `invokeAll()` method) is suspended until the tasks it sent to the pool finish their execution. This allows the `ForkJoinPool` class to use the work-stealing algorithm to assign a new task to the worker thread that executed the sleeping task. On the contrary, when you use the asynchronous methods (for example, the `fork()` method), the task continues with its execution, so the `ForkJoinPool` class can't use the work-stealing algorithm to increase the performance of the application. In this case, only when you call the `join()` or `get()` methods to wait for the finalization of a task, the `ForkJoinPool` class can use that algorithm.

In this recipe, you will learn how to use the asynchronous methods provided by the `ForkJoinPool` and `ForkJoinTask` classes for the management of tasks. You are going to implement a program that will search for files with a determined extension inside a folder and its subfolders. The `ForkJoinTask` class you're going to implement will process the content of a folder. For each subfolder inside that folder, it will send a new task to the `ForkJoinPool` class in an asynchronous way. For each file inside that folder, the task will check the extension of the file and add it to the result list if it proceeds.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `FolderProcessor` and specify that it extends the `RecursiveTask` class parameterized with the `List<String>` type.

   ```
   public class FolderProcessor extends RecursiveTask<List<String>> {
   ```

2. Declare the serial version UID of the class. This element is necessary because the parent class of the `RecursiveTask` class, the `ForkJoinTask` class, implements the `Serializable` interface.

   ```
     private static final long serialVersionUID = 1L;
   ```

3. Declare a private `String` attribute named `path`. This attribute will store the full path of the folder this task is going to process.

   ```
     private String path;
   ```

4. Declare a private `String` attribute named `extension`. This attribute will store the name of the extension of the files this task is going to look for.

   ```
     private String extension;
   ```

5. Implement the constructor of the class to initialize its attributes.

   ```
     public FolderProcessor (String path, String extension) {
       this.path=path;
       this.extension=extension;
     }
   ```

6. Implement the `compute()` method. As you parameterized the `RecursiveTask` class with the `List<String>` type, this method has to return an object of that type.

   ```
     @Override
     protected List<String> compute() {
   ```

7. Declare a list of `String` objects to store the names of the files stored in the folder.

   ```
       List<String> list=new ArrayList<>();
   ```

8. Declare a list of `FolderProcessor` tasks to store the subtasks that are going to process the subfolders stored in the folder.

   ```
       List<FolderProcessor> tasks=new ArrayList<>();
   ```

9. Get the content of the folder.

   ```
       File file=new File(path);
       File content[] = file.listFiles();
   ```

10. For each element in the folder, if there is a subfolder, create a new `FolderProcessor` object and execute it asynchronously using the `fork()` method.

```
    if (content != null) {
      for (int i = 0; i < content.length; i++) {
        if (content[i].isDirectory()) {
          FolderProcessor task=new FolderProcessor(content[i].
getAbsolutePath(), extension);
          task.fork();
          tasks.add(task);
```

11. Otherwise, compare the extension of the file with the extension you are looking for using the `checkFile()` method and, if they are equal, store the full path of the file in the list of strings declared earlier.

```
        } else {
          if (checkFile(content[i].getName())){
            list.add(content[i].getAbsolutePath());
          }
        }
      }
```

12. If the list of the `FolderProcessor` subtasks has more than 50 elements, write a message to the console to indicate this circumstance.

```
      if (tasks.size()>50) {
        System.out.printf("%s: %d tasks ran.\n",file.
getAbsolutePath(),tasks.size());
      }
```

13. Call the auxiliary method `addResultsFromTask()` that will add to the list of files the results returned by the subtasks launched by this task. Pass to it as parameters the list of strings and the list of the `FolderProcessor` subtasks.

```
      addResultsFromTasks(list,tasks);
```

14. Return the list of strings.

```
    return list;
```

15. Implement the `addResultsFromTasks()` method. For each task stored in the list of tasks, call the `join()` method that will wait for its finalization and then will return the result of the task. Add that result to the list of strings using the `addAll()` method.

```
  private void addResultsFromTasks(List<String> list,
      List<FolderProcessor> tasks) {
    for (FolderProcessor item: tasks) {
      list.addAll(item.join());
    }
  }
```

16. Implement the `checkFile()` method. This method compares if the name of a file passed as a parameter ends with the extension you are looking for. If so, the method returns the `true` value, otherwise it returns the `false` value.

```
  private boolean checkFile(String name) {
     return name.endsWith(extension);
  }
```

17. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

18. Create `ForkJoinPool` using the default constructor.

```
    ForkJoinPool pool=new ForkJoinPool();
```

19. Create three `FolderProcessor` tasks. Initialize each one with a different folder path.

```
    FolderProcessor system=new FolderProcessor("C:\\Windows",
"log");
    FolderProcessor apps=new
FolderProcessor("C:\\Program Files","log");
    FolderProcessor documents=new FolderProcessor("C:\\Documents
And Settings","log");
```

20. Execute the three tasks in the pool using the `execute()` method.

```
    pool.execute(system);
    pool.execute(apps);
    pool.execute(documents);
```

21. Write to the console information about the status of the pool every second until the three tasks have finished their execution.

```
    do {
      System.out.printf("****************************************
*\n");
      System.out.printf("Main: Parallelism: %d\n",pool.
getParallelism());
      System.out.printf("Main: Active Threads: %d\n",pool.
getActiveThreadCount());
      System.out.printf("Main: Task Count: %d\n",pool.
getQueuedTaskCount());
      System.out.printf("Main: Steal Count: %d\n",pool.
getStealCount());
      System.out.printf("****************************************
*\n");
      try {
```

```
            TimeUnit.SECONDS.sleep(1);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        } while ((!system.isDone())||(!apps.isDone())||(!documents.
    isDone()));
```

22. Shut down `ForkJoinPool` using the `shutdown()` method.

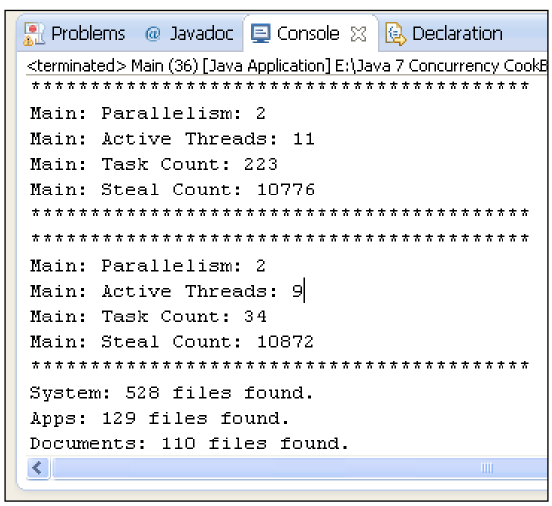
```
        pool.shutdown();
```

23. Write the number of results generated by each task to the console.

```
        List<String> results;

        results=system.join();
        System.out.printf("System: %d files found.\n",results.size());

        results=apps.join();
        System.out.printf("Apps: %d files found.\n",results.size());

        results=documents.join();
        System.out.printf("Documents: %d files found.\n",results.
    size());
```

## How it works...

The following screenshot shows part of an execution of this example:



The key of this example is in the `FolderProcessor` class. Each task processes the content of a folder. As you know, this content has the following two kinds of elements:

- Files
- Other folders

If the task finds a folder, it creates another `Task` object to process that folder and sends it to the pool using the `fork()` method. This method sends the task to the pool that will execute it if it has a free worker-thread or it can create a new one. The method returns immediately, so the task can continue processing the content of the folder. For every file, a task compares its extension with the one it's looking for and, if they are equal, adds the name of the file to the list of results.

Once the task has processed all the content of the assigned folder, it waits for the finalization of all the tasks it sent to the pool using the `join()` method. This method called in a task waits for the finalization of its execution and returns the value returned by the `compute()` method. The task groups the results of all the tasks it sent with its own results and returns that list as a return value of the `compute()` method.

The `ForkJoinPool` class also allows the execution of tasks in an asynchronous way. You have used the `execute()` method to send the three initial tasks to the pool. In the `Main` class, you also finished the pool using the `shutdown()` method and wrote information about the status and the evolution of the tasks that are running in it. The `ForkJoinPool` class includes more methods that can be useful for this purpose. See the *Monitoring a Fork/Join pool* recipe to see a complete list of those methods.

## There's more...

In this example, you have used the `join()` method to wait for the finalization of tasks and get their results. You can also use one of the two versions of the `get()` method with this purpose:

- `get()`: This version of the `get()` method returns the value returned by the `compute()` method if `ForkJoinTask` has finished its execution, or waits until its finalization.
- `get(long timeout, TimeUnit unit)`: This version of the `get()` method, if the result of the task isn't available, waits the specified time for it. If the specified period of time passes and the result isn't yet available, the method returns a `null` value. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

There are two main differences between the `get()` and the `join()` methods:

- The `join()` method can't be interrupted. If you interrupt the thread that called the `join()` method, the method throws an `InterruptedException` exception.
- While the `get()` method will return an `ExecutionException` exception if the tasks throw any unchecked exception, the `join()` method will return a `RuntimeException` exception.

## See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*
- The *Monitoring a Fork/Join pool* recipe in *Chapter 8*, *Testing concurrent applications*

# Throwing exceptions in the tasks

There are two kinds of exceptions in Java:

- **Checked exceptions**: These exceptions must be specified in the `throws` clause of a method or caught inside them. For example, `IOException` or `ClassNotFoundException`.
- **Unchecked exceptions**: These exceptions don't have to be specified or caught. For example, `NumberFormatException`.

You can't throw any checked exception in the `compute()` method of the `ForkJoinTask` class, because this method doesn't include any throws declaration in its implementation. You have to include the necessary code to handle exceptions. On the other hand, you can throw (or it can be thrown by any method or object used inside the method) an unchecked exception. The behavior of the `ForkJoinTask` and `ForkJoinPool` classes is different from what you may expect. The program doesn't finish execution and you won't see any information about the exception in the console. It's simply swallowed as if it weren't thrown. You can, however, use some methods of the `ForkJoinTask` class to know if a task threw an exception and what kind of exception it was. In this recipe, you will learn how to get that information.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task`. Specify that it implements the `RecursiveTask` class parameterized with the `Integer` class.

   ```
   public class Task extends RecursiveTask<Integer> {
   ```

2. Declare a private `int` array named `array`. It will simulate the array of data you are going to process in this example.

   ```
     private int array[];
   ```

3. Declare two private `int` attributes named `start` and `end`. These attributes will determine the elements of the array this task has to process.

   ```
     private int start, end;
   ```

4. Implement the constructor of the class that initializes its attributes.

   ```
     public Task(int array[], int start, int end){
       this.array=array;
       this.start=start;
       this.end=end;
     }
   ```

5. Implement the `compute()` method of the task. As you parameterized the `RecursiveTask` class with the `Integer` class, this method has to return an `Integer` object. First, write a message to the console with the value of the `start` and `end` attributes.

   ```
     @Override
     protected Integer compute() {
       System.out.printf("Task: Start from %d to %d\n",start,end);
   ```

6. If the block of elements that this task has to process, determined by the `start` and `end` attributes, has a size smaller than 10, check if the element in the fourth position in the array (index number three) is in that block. If that is the case, throw a `RuntimeException` exception. Then, put the task to sleep for a second.

   ```
       if (end-start<10) {
         if ((3>start)&&(3<end)){
           throw new RuntimeException("This task throws an"+
   "Exception: Task from  "+start+" to "+end);
         }
         try {
           TimeUnit.SECONDS.sleep(1);
         } catch (InterruptedException e) {
           e.printStackTrace();
         }
   ```

7. Otherwise (the block of elements that this task has to process has a size of 10 or bigger), divide the block of elements in two, create two `Task` objects to process those blocks, and execute them in the pool using the `invokeAll()` method.

```
} else {
  int mid=(end+start)/2;
  Task task1=new Task(array,start,mid);
  Task task2=new Task(array,mid,end);
  invokeAll(task1, task2);
}
```

8. Write a message to the console indicating the end of the task writing the value of the `start` and `end` attributes.

```
System.out.printf("Task: End form %d to %d\n",start,end);
```

9. Return the number `0` as result of the task.

```
return 0;
```

10. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

11. Create an array of 100 integer numbers.

```
int array[]=new int[100];
```

12. Create a `Task` object to process that array.

```
Task task=new Task(array,0,100);
```

13. Create a `ForkJoinPool` object using the default constructor.

```
ForkJoinPool pool=new ForkJoinPool();
```

14. Execute the task in the pool using the `execute()` method.

```
pool.execute(task);
```

15. Shut down the `ForkJoinPool` class using the `shutdown()` method.

```
pool.shutdown();
```

16. Wait for the finalization of the task using the `awaitTermination()` method. As you want to wait for the finalization of the task however long it takes to complete, pass the values `1` and `TimeUnit.DAYS` as parameters to this method.

```
try {
  pool.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

17. Check if the task, or one of its subtasks, has thrown an exception using the `isCompletedAbnormally()` method. In that case, write a message to the console with the exception that was thrown. Get that exception with the `getException()` method of the `ForkJoinTask` class.

```
if (task.isCompletedAbnormally()) {
  System.out.printf("Main: An exception has ocurred\n");
  System.out.printf("Main: %s\n",task.getException());
}
System.out.printf("Main: Result: %d",task.join());
```

## How it works...

The `Task` class you have implemented in this recipe processes an array of numbers. It checks if the block of numbers it has to process has 10 or more elements. In that case, it splits the block in two and creates two new `Task` objects to process those blocks. Otherwise, it looks for the element in the fourth position of the array (index number three). If that element is in the block the task has to process, it throws a `RuntimeException` exception.

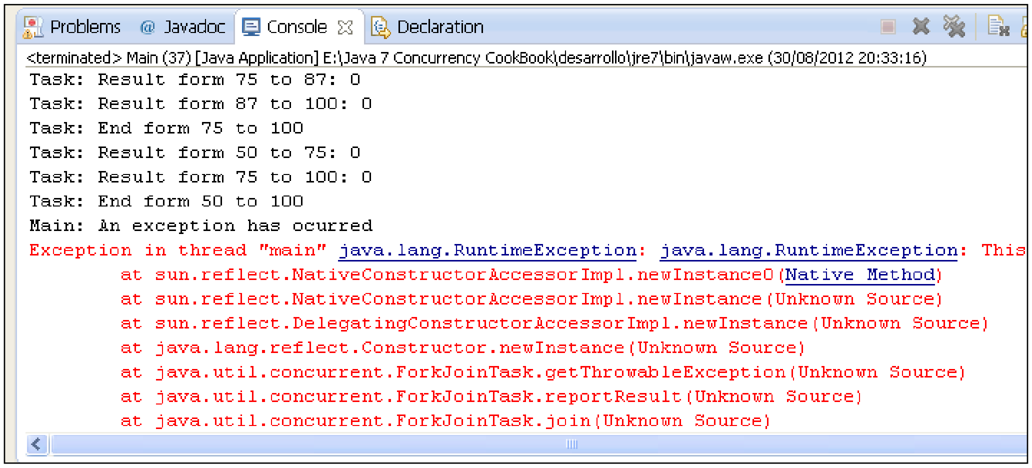
When you execute the program, the exception is thrown, but the program doesn't stop. In the `Main` class you have included a call to the `isCompletedAbnormally()` method of the `ForkJoinTask` class using the original task. This method returns `true` if that task, or one of its subtasks, has thrown an exception. You also used the `getException()` method of the same object to get the `Exception` object that it has thrown.

When you throw an unchecked exception in a task, it also affects its parent task (the task that sent it to the `ForkJoinPool` class) and the parent task of its parent task, and so on. If you revise all the output of the program, you'll see that there aren't output messages for the finalization of some tasks. The stating messages of those tasks are as follows:

```
Task: Starting form 0 to 100
Task: Starting form 0 to 50
Task: Starting form 0 to 25
Task: Starting form 0 to 12
Task: Starting form 0 to 6
```

These tasks are the ones that threw the exception and its parent tasks. All of them have finished abnormally. Take this into account, when you develop a program with the `ForkJoinPool` and `ForkJoinTask` objects that can throw exceptions if you don't want this behavior.

The following screenshot shows part of an execution of this example:




## There's more...

You can obtain the same result obtained in the example, if instead of throwing an exception, you use the `completeExceptionally()` method of the `ForkJoinTask` class. The code would be like the following:

```
Exception e=new Exception("This task throws an Exception: "+ "Task
from  "+start+" to "+end);
completeExceptionally(e);
```

## See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*

# Canceling a task

When you execute the `ForkJoinTask` objects in a `ForkJoinPool` class, you can cancel them before they start their execution. The `ForkJoinTask` class provides the `cancel()` method for this purpose. There are some points you have to take into account when you want to cancel a task, which are as follows:

- The `ForkJoinPool` class doesn't provide any method to cancel all the tasks it has running or waiting in the pool
- When you cancel a task, you don't cancel the tasks this task has executed

In this recipe, you will implement an example of cancelation of `ForkJoinTask` objects. You will look for the position of a number in an array. The first task that finds the number will cancel the remaining tasks. As that functionality is not provided by the Fork/Join framework, you will implement an auxiliary class to do this cancelation.

## Getting ready...

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE NetBeans, open it and create a new Java project

## How to do it...

Follow these steps to implement the example:

1. Create a class named `ArrayGenerator`. This class will generate an array of random integer numbers with the specified size. Implement a method named `generateArray()`. It will generate the array of numbers. It receives the size of the array as a parameter.

```
public class ArrayGenerator {
  public int[] generateArray(int size) {
    int array[]=new int[size];
    Random random=new Random();
    for (int i=0; i<size; i++){
      array[i]=random.nextInt(10);
    }
    return array;
  }
```

2. Create a class named `TaskManager`. We will use this class to store all the tasks executed in `ForkJoinPool` used in the example. Due to the limitations of the `ForkJoinPool` and `ForkJoinTask` classes, you will use this class to cancel all the tasks of the `ForkJoinPool` class.

```
public class TaskManager {
```

3. Declare a list of objects parameterized with the `ForkJoinTask` class parameterized with the `Integer` class named `List`.

```
  private List<ForkJoinTask<Integer>> tasks;
```

4. Implement the constructor of the class. It initializes the list of tasks.

```
  public TaskManager(){
    tasks=new ArrayList<>();
  }
```

5. Implement the `addTask()` method. It adds a `ForkJoinTask` object to the lists of tasks.

```
public void addTask(ForkJoinTask<Integer> task){
  tasks.add(task);
}
```

6. Implement the `cancelTasks()` method. It will cancel all the `ForkJoinTask` objects stored in the list using the `cancel()` method. It receives as a parameter the `ForkJoinTask` object that wants to cancel the rest of the tasks. The method cancels all the tasks.

```
public void cancelTasks(ForkJoinTask<Integer> cancelTask){
  for (ForkJoinTask<Integer> task  :tasks) {
    if (task!=cancelTask) {
      task.cancel(true);
      ((SearchNumberTask)task).writeCancelMessage();
    }
  }
}
```

7. Implement the `SearchNumberTask` class. Specify that it extends the `RecursiveTask` class parameterized with the `Integer` class. This class will look for a number in a block of elements of an integer array.

```
public class SearchNumberTask extends RecursiveTask<Integer> {
```

8. Declare a private array of `int` numbers named `array`.

```
private int numbers[];
```

9. Declare two private `int` attributes named `start` and `end`. These attributes will determine the elements of the array this task has to process.

```
private int start, end;
```

10. Declare a private `int` attribute named `number` to store the number you are going to look for.

```
private int number;
```

11. Declare a private `TaskManager` attribute named `manager`. You will use this object to cancel all the tasks.

```
private TaskManager manager;
```

12. Declare a private `int` constant and initialize it to the `-1` value. It will be the returned value by the task when it doesn't find the number.

```
private final static int NOT_FOUND=-1;
```

13. Implement the constructor of the class to initialize its attributes.

```
  public Task(int numbers[], int start, int end, int number,
TaskManager manager){
    this.numbers=numbers;
    this.start=start;
    this.end=end;
    this.number=number;
    this.manager=manager;
  }
```

14. Implement the `compute()` method. Start the method by writing a message to the console indicating the values of the `start` and `end` attributes.

```
  @Override
  protected Integer compute() {
    System.out.println("Task: "+start+":"+end);
```

15. If the difference between the `start` and `end` attributes are bigger than 10 (the task has to process more than 10 elements of the array), call the `launchTasks()` method to divide the work of this task in two subtasks.

```
    int ret;
    if (end-start>10) {
      ret=launchTasks();
```

16. Otherwise, look for the number in the block of the array this task that is calling the `lookForNumber()` method has to process.

```
    } else {
      ret=lookForNumber();
    }
```

17. Return the result of the task.

```
    return ret;
```

18. Implement the `lookForNumber()` method.

```
  private int lookForNumber() {
```

19. For all the elements in the block of elements this task has to process, compare the value stored in that element with the number you are looking for. If they are equal, write a message to the console indicating that in such a circumstance use the `cancelTasks()` method of the `TaskManager` object to cancel all the tasks, and return the position of the element, where you found the number.

```
    for (int i=start; i<end; i++){
      if (array[i]==number) {
        System.out.printf("Task: Number %d found in position
%d\n",number,i);
        manager.cancelTasks(this);
```

```
        return i;
      }
```

20. Inside the loop, put the task to sleep for one second.

```
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

21. Finally, return the `-1` value.

```
    return NOT_FOUND;
  }
```

22. Implement the `launchTasks()` method. First, divide the block of numbers this tasks has to process in two and then, create two `Task` objects to process them.

```
  private int launchTasks() {
    int mid=(start+end)/2;

    Task task1=new Task(array,start,mid,number,manager);
    Task task2=new Task(array,mid,end,number,manager);
```

23. Add the tasks to the `TaskManager` object.

```
    manager.addTask(task1);
    manager.addTask(task2);
```

24. Execute the two tasks asynchronously using the `fork()` method.

```
    task1.fork();
    task2.fork();
```

25. Wait for the finalization of the tasks and return the result of the first task if it is different, to `-1`, or the result of the second task.

```
    int returnValue;

    returnValue=task1.join();
    if (returnValue!=-1) {
      return returnValue;
    }

    returnValue=task2.join();
    return returnValue;
```

26. Implement the `writeCancelMessage()` method to write a message when the task is canceled.

```
  public void writeCancelMessage(){
    System.out.printf("Task: Canceled task from %d to
%d",start,end);
  }
```

27. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

28. Create an array of 1,000 numbers using the `ArrayGenerator` class.

```
    ArrayGenerator generator=new ArrayGenerator();
    int array[]=generator.generateArray(1000);
```

29. Create a `TaskManager` object.

```
    TaskManager manager=new TaskManager();
```

30. Create a `ForkJoinPool` object using the default constructor.

```
    ForkJoinPool pool=new ForkJoinPool();
```

31. Create a `Task` object to process the array generated before.

```
    Task task=new Task(array,0,1000,5,manager);
```

32. Execute the task in the pool asynchronously using the `execute()` method.

```
    pool.execute(task);
```

33. Shut down the pool using the `shutdown()` method.

```
    pool.shutdown();
```

34. Wait for the finalization of the tasks using the `awaitTermination()` method of the `ForkJoinPool` class.

```
    try {
      pool.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

35. Write a message to the console indicating the end of the program.

```
    System.out.printf("Main: The program has finished\n");
```

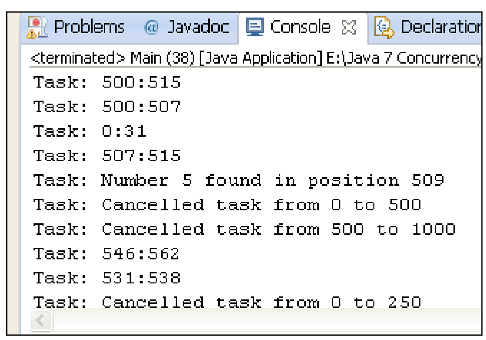
## How it works...

The `ForkJoinTask` class provides the `cancel()` method that allows you to cancel a task if it hasn't been executed yet. This is a very important point. If the task has begun its execution, a call to the `cancel()` method has no effect. The method receives a parameter as a `Boolean` value called `mayInterruptIfRunning`. This name may make you think that, if you pass the `true` value to the method, the task will be canceled even if it is running. The Java API documentation specifies that, in the default implementation of the `ForkJoinTask` class, this attribute has no effect. The tasks are only canceled if they haven't started their execution. The cancelation of a task has no effect over the tasks that this task sent to the pool. They continue with their execution.

A limitation of the Fork/Join framework is that it doesn't allow the cancelation of all the tasks that are in `ForkJoinPool`. To overcome that limitation, you have implemented the `TaskManager` class. It stores all the tasks that have been sent to the pool. It has a method that cancels all the tasks it has stored. If a task can't be canceled because it's running or it has finished, the `cancel()` method returns the `false` value, so you can try to cancel all the tasks without being afraid of possible collateral effects.

In the example, you have implemented a task that looks for a number in an array of numbers. You divide the problem into smaller sub-problems as the Fork/Join framework recommends. You are only interested in one occurrence of the number so, when you find it, you cancel the other tasks.

The following screenshot shows part of an execution of this example:




## See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*

# 6

# Concurrent Collections

In this chapter we will cover:

- Using non-blocking thread-safe lists
- Using blocking thread-safe lists
- Using blocking thread-safe lists ordered by priority
- Using thread-safe lists with delayed elements
- Using thread-safe navigable maps
- Generating concurrent random numbers
- Using atomic variables
- Using atomic arrays

## Introduction

**Data structures** are a basic element in programming. Almost every program uses one or more types of data structures to store and manage their data. Java API provides the **Java Collections framework** that contains interfaces, classes, and algorithms, which implement a lot of different data structures that you can use in your programs.

When you need to work with data collections in a concurrent program, you must be very careful with the implementation you choose. Most collection classes are not ready to work with concurrent applications because they don't control the concurrent access to its data. If some concurrent tasks share a data structure that is not ready to work with concurrent tasks, you can have data inconsistency errors that will affect the correct operation of the program. One example of this kind of data structures is the `ArrayList` class.

Java provides data collections that you can use in your concurrent programs without any problems or inconsistency. Basically, Java provides two kinds of collections to use in concurrent applications:

- **Blocking collections**: This kind of collection includes operations to add and remove data. If the operation can't be made immediately, because the collection is full or empty, the thread that makes the call will be blocked until the operation can be made.
- **Non-blocking collections**: This kind of collection also includes operations to add and remove data. If the operation can't be made immediately, the operation returns a `null` value or throws an exception, but the thread that makes the call won't be blocked.

Through the recipes of this chapter, you will learn how to use some Java collections that you can use in your concurrent applications. This includes:

- Non-blocking lists, using the `ConcurrentLinkedDeque` class
- Blocking lists, using the `LinkedBlockingDeque` class
- Blocking lists to be used with producers and consumers of data, using the `LinkedTransferQueue` class
- Blocking lists that order their elements by priority, with the `PriorityBlockingQueue`
- Blocking lists with delayed elements, using the `DelayQueue` class
- Non-blocking navigable maps, using the `ConcurrentSkipListMap` class
- Random numbers, using the `ThreadLocalRandom` class
- Atomic variables, using the `AtomicLong` and `AtomicIntegerArray` classes

# Using non-blocking thread-safe lists

The most basic collection is the **list**. A list has an undetermined number of elements and you can add, read, or remove the element of any position. Concurrent lists allow the various threads to add or remove elements in the list at a time without producing any data inconsistency.

In this recipe, you will learn how to use non-blocking lists in your concurrent programs. Non-blocking lists provide operations that, if the operation can't be done immediately (for example, you want to get an element of the list and the list is empty), they throw an exception or return a `null` value, depending on the operation. Java 7 has introduced the `ConcurrentLinkedDeque` class that implements a non-blocking concurrent list.

We are going to implement an example with the following two different tasks:

- One that adds data to a list massively
- One that removes data from the same list massively

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `AddTask` and specify that it implements the `Runnable` interface.

   ```
   public class AddTask implements Runnable {
   ```

2. Declare a private `ConcurrentLinkedDeque` attribute parameterized with the `String` class named `list`.

   ```
     private ConcurrentLinkedDeque<String> list;
   ```

3. Implement the constructor of the class to initialize its attribute.

   ```
     public AddTask(ConcurrentLinkedDeque<String> list) {
       this.list=list;
     }
   ```

4. Implement the `run()` method of the class. It will store 10,000 strings in the list with the name of the thread that is executing the task and a number.

   ```
      @Override
     public void run() {
       String name=Thread.currentThread().getName();
       for (int i=0; i<10000; i++){
         list.add(name+": Element "+i);
       }
     }
   ```

5. Create a class named `PollTask` and specify that it implements the `Runnable` interface.

   ```
   public class PollTask implements Runnable {
   ```

6. Declare a private `ConcurrentLinkedDeque` attribute parameterized with the `String` class named `list`.

```
private ConcurrentLinkedDeque<String> list;
```

7. Implement the constructor of the class to initialize its attribute.

```
public PollTask(ConcurrentLinkedDeque<String> list) {
  this.list=list;
}
```

8. Implement the `run()` method of the class. It takes out 10,000 elements of the list in a loop with 5,000 steps, taking off two elements in each step.

```
 @Override
public void run() {
  for (int i=0; i<5000; i++) {
    list.pollFirst();
    list.pollLast();
  }
}
```

9. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

10. Create a `ConcurrentLinkedDeque` object parameterized with the `String` class named `list`.

```
    ConcurrentLinkedDeque<String> list=new
ConcurrentLinkedDeque<>();
```

11. Create an array for 100 `Thread` objects named `threads`.

```
    Thread threads[]=new Thread[100];
```

12. Create 100 `AddTask` objects and a thread to run each of them. Store every thread in the array created earlier and start the threads.

```
    for (int i=0; i<threads.length ; i++){
      AddTask task=new AddTask(list);
      threads[i]=new Thread(task);
      threads[i].start();
    }
    System.out.printf("Main: %d AddTask threads have been
launched\n",threads.length);
```

13. Wait for the completion of the threads using the `join()` method.

```
    for (int i=0; i<threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

14. Write in the console the size of the list.

```
    System.out.printf("Main: Size of the List: %d\n",list.size());
```

15. Create 100 `PollTask` objects and a thread to run each of them. Store every thread in the array created earlier and start the threads.

```
    for (int i=0; i< threads.length; i++){
      PollTask task=new PollTask(list);
      threads[i]=new Thread(task);
      threads[i].start();
    }
    System.out.printf("Main: %d PollTask threads have been
launched\n",threads.length);
```

16. Wait for the finalization of the threads using the `join()` method.

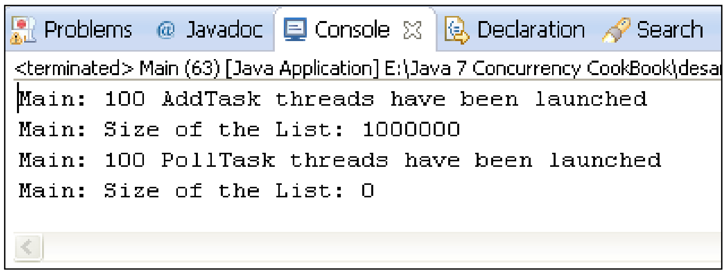
```
    for (int i=0; i<threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

17. Write in the console the size of the list.

```
    System.out.printf("Main: Size of the List: %d\n",list.size());
```

## How it works...

In this recipe, we have used the `ConcurrentLinkedDeque` object parameterized with the `String` class to work with a non-blocking concurrent list of data. The following screenshot shows the output of an execution of this example:




First, you have executed 100 `AddTask` tasks to add elements to the list. Each one of those tasks inserts 10,000 elements to the list using the `add()` method. This method adds the new elements at the end of the list. When all those tasks have finished, you have written in the console the number of elements of the list. At this moment, the list has 1,000,000 elements.

Then, you have executed 100 `PollTask` tasks to remove elements from the list. Each one of those tasks removes 10,000 elements of the list using the `pollFirst()` and `pollLast()` methods. The `pollFirst()` method returns and removes the first element of the list and the `pollLast()` method returns and removes the last element of the list. If the list is empty, these methods return a `null` value. When all those tasks have finished, you have written in the console the number of elements of the list. At this moment, the list has zero elements.

To write the number of elements of the list, you have used the `size()` method. You have to take into account that this method can return a value that is not real, especially if you use it when there are threads adding or deleting data in the list. The method has to traverse the entire list to count the elements and the contents of the list can change for this operation. Only if you use them when there aren't any threads modifying the list, you will have the guarantee that the returned result is correct.

## There's more...

The `ConcurrentLinkedDeque` class provides more methods to get elements form the list:

- `getFirst()` and `getLast()`: These methods return the first and last element from the list respectively. They don't remove the returned element from the list. If the list is empty, these methods throw a `NoSuchElementExcpetion` exception.
- `peek()`, `peekFirst()`, and `peekLast()`: These methods return the first and the last element of the list respectively. They don't remove the returned element from the list. If the list is empty, these methods return a `null` value.

- `remove()`, `removeFirst()`, `removeLast()`: These methods return the first and the last element of the list respectively. They remove the returned element from the list. If the list is empty, these methods throw a `NoSuchElementException` exception.

# Using blocking thread-safe lists

The most basic collection is the list. A list has an undetermined number of elements and you can add, read, or remove the element from any position. A concurrent list allows various threads to add or remove elements in the list at a time without producing any data inconsistency.

In this recipe, you will learn how to use blocking lists in your concurrent programs. The main difference between blocking lists and non-blocking lists is that blocking lists has methods to insert and delete elements on it that, if they can't do the operation immediately, because the list is full or empty, they block the thread that make the call until the operation can be made. Java includes the `LinkedBlockingDeque` class that implements a blocking list.

You are going to implement an example with the following two tasks:

- One that adds data to a list massively
- One that removes data from the same list massively

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow the steps described next to implement the example:

1. Create a class named `Client` and specify that it implements the `Runnable` interface.

   ```
   public class Client implements Runnable{
   ```

2. Declare a private `LinkedBlockingDeque` attribute parameterized with the `String` class named `requestList`.

   ```
     private LinkedBlockingDeque<String> requestList;
   ```

3. Implement the constructor of the class to initialize its attributes.

   ```
     public Client (LinkedBlockingDeque<String> requestList) {
       this.requestList=requestList;
     }
   ```

4. Implement the `run()` method. Insert five `String` objects in the list per second using the `put()` method of `requestList object`. Repeat that cycle three times.

```
  @Override
  public void run() {
    for (int i=0; i<3; i++) {
      for (int j=0; j<5; j++) {
        StringBuilder request=new StringBuilder();
        request.append(i);
        request.append(":");
        request.append(j);
        try {
          requestList.put(request.toString());
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.printf("Client: %s at %s.\n",request,new
Date());
      }
      try {
        TimeUnit.SECONDS.sleep(2);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.printf("Client: End.\n");
  }
```

5. Create the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

6. Declare and create `LinkedBlockingDeque` parameterized with the `String` class named `list`.

```
    LinkedBlockingDeque<String> list=new LinkedBlockingDeque<>(3);
```

7. Create and start a `Thread` object to execute a client task.

```
    Client client=new Client(list);
    Thread thread=new Thread(client);
    thread.start();
```

8. Get three `String` objects of the list every 300 milliseconds using the `take()` method of the list object. Repeat that cycle five times. Write the strings in the console.

```
    for (int i=0; i<5 ; i++) {
      for (int j=0; j<3; j++) {
        String request=list.take();
        System.out.printf("Main: Request: %s at %s. Size:
%d\n",request,new Date(),list.size());
      }
      TimeUnit.MILLISECONDS.sleep(300);
    }
```

9. Write a message to indicate the end of the program.

```
    System.out.printf("Main: End of the program.\n");
```

## How it works...

In this recipe, you have used `LinkedBlockingDeque` parameterized with the `String` class to work with a non-blocking concurrent list of data.

The `Client` class uses the `put()` method to insert strings in the list. If the list is full (because you have created it with a fixed capacity), the method blocks the execution of its thread until there is an empty space in the list.

The `Main` class uses the `take()` method to get strings from the list. If the list is empty, the method blocks the execution of its thread until there are elements in the list.

Both the methods of the `LinkedBlockingDeque` class used in this example, can throw an `InterruptedException` exception if they are interrupted while they are blocked, so you have to include the necessary code to catch that exception.

## There's more...

The `LinkedBlockingDeque` class also provides the methods to put and get elements from the list that, instead of being block, throw an exception or return the `null` value. These methods are:

- `takeFirst()` and `takeLast()`: These methods return the first and the last element of the list respectively. They remove the returned element from the list. If the list is empty, these methods block the thread until there are elements in the list.
- `getFirst()` and `getLast()`: These methods return the first and last element from the list respectively. They don't remove the returned element from the list. If the list is empty, these methods throw a `NoSuchElementExcpetion` exception.

- `peek()`, `peekFirst()`, and `peekLast()`: These methods return the first and the last element of the list respectively. They don't remove the returned element from the list. If the list is empty, these methods return a `null` value.
- `poll()`, `pollFirst()`, and `pollLast()`: These methods return the first and the last element of the list respectively. They remove the returned element form the list. If the list is empty, these methods return a `null` value.
- `add()`, `addFirst()`, `addLast()`: These methods add an element in the first and the last position respectively. If the list is full (you have created it with a fixed capacity), these methods throw an `IllegalStateException` exception.

## See also

- The *Using non-blocking thread-safe lists* recipe in *Chapter 6*, *Concurrent Collections*

# Using blocking thread-safe lists ordered by priority

A typical need when you work with data structures is to have an ordered list. Java provides `PriorityBlockingQueue` that has this functionality.

All the elements you want to add to `PriorityBlockingQueue` have to implement the `Comparable` interface. This interface has a method, `compareTo()` that receives an object of the same type, so you have two objects to compare: the one that is executing the method and the one that is received as a parameter. The method must return a number less than zero if the local object is less than the parameter, a number bigger that zero if the local object is greater than the parameter, and the number zero if both objects are equal.

`PriorityBlockingQueue` uses the `compareTo()` method when you insert an element in it to determine the position of the element inserted. The greater elements will be the tail of the queue.

Another important characteristic of `PriorityBlockingQueue` is that it's a **blocking data structure**. It has methods that, if they can't do their operation immediately, block the thread until they can do it.

In this recipe, you will learn how to use the `PriorityBlockingQueue` class implementing an example, where you are going to store a lot of events with different priorities in the same list, to check that the queue will be ordered as you want.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Event` and specify that it implements the `Comparable` interface parameterized with the `Event` class.

   ```
   public class Event implements Comparable<Event> {
   ```

2. Declare a private `int` attribute named `thread` to store the number of the thread that has created the event.

   ```
     private int thread;
   ```

3. Declare a private `int` attribute named `priority` to store the priority of the event.

   ```
     private int priority;
   ```

4. Implement the constructor of the class to initialize its attributes.

   ```
     public Event(int thread, int priority){
       this.thread=thread;
       this.priority=priority;
     }
   ```

5. Implement the `getThread()` method to return the value of the thread attribute.

   ```
     public int getThread() {
       return thread;
     }
   ```

6. Implement the `getPriority()` method to return the value of the priority attribute.

   ```
     public int getPriority() {
       return priority;
     }
   ```

7. Implement the `compareTo()` method. It receives `Event` as a parameter and compares the priority of the current event and the one received as parameter. It returns `-1` if the priority of the current event is bigger, `0` if both priorities are equal, and `1` if the priority of the current event is smaller. Note that this is opposite of most `Comparator.compareTo()` implementations.

   ```
   @Override
     public int compareTo(Event e) {
       if (this.priority>e.getPriority()) {
         return -1;
       } else if (this.priority<e.getPriority()) {
         return 1;
       } else {
         return 0;
       }
     }
   ```

8. Create a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable {
```

9. Declare a private `int` attribute named `id` to store the number that identifies the task.

```
  private int id;
```

10. Declare a private `PriorityBlockingQueue` attribute parameterized with the `Event` class named `queue` to store the events generated by the task.

```
  private PriorityBlockingQueue<Event> queue;
```

11. Implement the constructor of the class to initialize its attributes.

```
  public Task(int id, PriorityBlockingQueue<Event> queue) {
    this.id=id;
    this.queue=queue;
  }
```

12. Implement the `run()` method. It stores 1000 events in the queue, using its ID to identify the task that creates the event and giving to them as priority an increasing number. Use the `add()` method to store the events in the queue.

```
   @Override
  public void run() {
    for (int i=0; i<1000; i++){
      Event event=new Event(id,i);
      queue.add(event);
    }
  }
```

13. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main{
  public static void main(String[] args) {
```

14. Create a `PriorityBlockingQueue` object parameterized with the `Event` class named `queue`.

```
    PriorityBlockingQueue<Event> queue=new 
PriorityBlockingQueue<>();
```

15. Create an array of five `Thread` objects to store the threads that is going to execute five tasks.

```
    Thread taskThreads[]=new Thread[5];
```

16. Create five `Task` objects. Store the threads in the array created earlier.

```
    for (int i=0; i<taskThreads.length; i++){
      Task task=new Task(i,queue);
```

```
    taskThreads[i]=new Thread(task);
  }
```

17. Start the five threads created earlier.

```
  for (int i=0; i<taskThreads.length ; i++) {
    taskThreads[i].start();
  }
```

18. Wait for the finalization of the five threads using the `join()` method.

```
  for (int i=0; i<taskThreads.length ; i++) {
    try {
      taskThreads[i].join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
```

19. Write to the console the actual size of the queue and the events stored in it. Use the `poll()` method to take off the events from the queue.

```
  System.out.printf("Main: Queue Size: %d\n",queue.size());
  for (int i=0; i<taskThreads.length*1000; i++){
    Event event=queue.poll();
    System.out.printf("Thread %s: Priority %d\n",event.
getThread(),event.getPriority());
  }
```

20. Write a message to the console with the final size of the queue.

```
  System.out.printf("Main: Queue Size: %d\n",queue.size());
  System.out.printf("Main: End of the program\n");
```

## How it works...

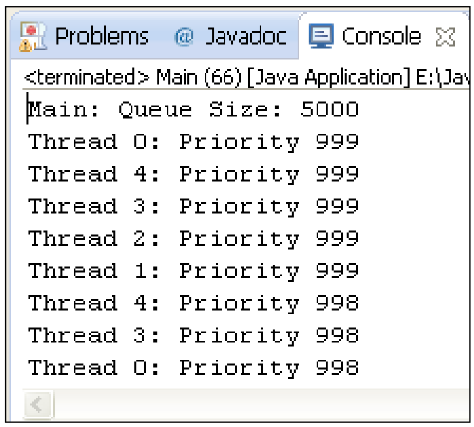
In this example, you have implemented a priority queue of `Event` objects using `PriorityBlockingQueue`. As we mentioned in the introduction, all the elements stored in `PriorityBlockingQueue` have to implement the `Comparable` interface, so you have implemented the `compareTo()` method in the Event class.

All the events have a priority attribute. The elements that have a higher value of priority will be the first elements in the queue. When you have implemented the `compareTo()` method, if the event executing the method has a priority higher than the priority of the event passed as parameter, it returns `-1` as the result. In the other case, if the event executing the method has a priority lower than the priority of the event passed as parameter, it returns `1` as the result. If both objects have the same priority, the `compareTo()` method returns the `0` value. In that case, the `PriorityBlockingQueue` class doesn't guarantee the order of the elements.

We have implemented the `Task` class to add the `Event` objects to the priority queue. Each task object adds to the queue 1,000 events, with priorities between 0 and 999, using the `add()` method.

The `main()` method of the `Main` class creates five `Task` objects and executes them in the corresponding threads. When all the threads have finished their execution, you have written all the elements to the console. To get the elements from the queue, we have used the `poll()` method. That method returns and removes the first element from the queue.

The following screenshot shows part of the output of an execution of the program:




You can see how the queue has a size of 5,000 elements and how the first elements have the biggest priority values.

## There's more...

The `PriorityBlockingQueue` class has other interesting methods. Following is the description of some of them:

- `clear()`: This method removes all the elements of the queue.
- `take()`: This method returns and removes the first element of the queue. If the queue is empty, the method blocks its thread until the queue has elements.
- `put(E e)`: `E` is the class used to parameterize the `PriorityBlockingQueue` class. This method inserts the element passed as a parameter into the queue.
- `peek()`:This method returns the first element of the queue, but doesn't remove it.

## See also

- The *Using blocking thread-safe lists* recipe in *Chapter 6, Concurrent Collections*

# Using thread-safe lists with delayed elements

An interesting data structure provided by the Java API, and that you can use in concurrent applications, is implemented in the `DelayedQueue` class. In this class, you can store elements with an activation date. The methods that return or extract elements of the queue will ignore those elements whose data is in the future. They are invisible to those methods.

To obtain this behavior, the elements you want to store in the `DelayedQueue` class have to implement the `Delayed` interface. This interface allows you to work with delayed objects, so you will implement the activation date of the objects stored in the `DelayedQueue` class as the time remaining until the activation date. This interface forces to implement the following two methods:

- `compareTo(Delayed o)`: The `Delayed` interface extends the `Comparable` interface. This method will return a value less than zero if the object that is executing the method has a delay smaller than the object passed as a parameter, a value greater than zero if the object that is executing the method has a delay bigger than the object passed as a parameter, and the zero value if both objects have the same delay.
- `getDelay(TimeUnit unit)`: This method has to return the time remaining until the activation date in the units is specified by the unit parameter. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`.

In this example, you will learn how to use the `DelayedQueue` class storing in it some events with different activation dates.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Event` and specify that it implements the `Delayed` interface.

   ```
   public class Event implements Delayed {
   ```

2. Declare a private `Date` attribute named `startDate`.

   ```
     private Date startDate;
   ```

3. Implement the constructor of the class to initialize its attribute.

```
public Event (Date startDate) {
  this.startDate=startDate;
}
```

4. Implement the `compareTo()` method. It receives a `Delayed` object as its parameter. Return the difference between the delay of the current object and the one passed as parameter.

```
@Override
  public int compareTo(Delayed o) {
    long result=this.getDelay(TimeUnit.NANOSECONDS)-o.
getDelay(TimeUnit.NANOSECONDS);
    if (result<0) {
      return -1;
    } else if (result>0) {
      return 1;
    }
    return 0;
  }
```

5. Implement the `getDelay()` method. Return the difference between `startDate` of the object and the actual `Date` in `TimeUnit` received as parameter.

```
public long getDelay(TimeUnit unit) {
  Date now=new Date();
  long diff=startDate.getTime()-now.getTime();
  return unit.convert(diff,TimeUnit.MILLISECONDS);
}
```

6. Create a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable {
```

7. Declare a private `int` attribute named `id` to store a number that identifies this task.

```
private int id;
```

8. Declare a private `DelayQueue` attribute parameterized with the `Event` class named `queue`.

```
private DelayQueue<Event> queue;
```

9. Implement the constructor of the class to initialize its attributes.

```
public Task(int id, DelayQueue<Event> queue) {
  this.id=id;
  this.queue=queue;
}
```

10. Implement the `run()` method. First, calculate the activation date of the events that this task is going to create. Add to the actual date a number of seconds equal to the ID of the object.

```
@Override
  public void run() {
    Date now=new Date();
    Date delay=new Date();
    delay.setTime(now.getTime()+(id*1000));
    System.out.printf("Thread %s: %s\n",id,delay);
```

11. Store 100 events in the queue using the `add()` method.

```
    for (int i=0; i<100; i++) {
      Event event=new Event(delay);
      queue.add(event);
    }
  }
```

12. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) throws Exception {
```

13. Create a `DelayedQueue` object parameterized with the `Event` class.

```
    DelayQueue<Event> queue=new DelayQueue<>();
```

14. Create an array of five `Thread` objects to store the tasks you're going to execute.

```
    Thread threads[]=new Thread[5];
```

15. Create five `Task` objects, with different IDs.

```
    for (int i=0; i<threads.length; i++){
      Task task=new Task(i+1, queue);
      threads[i]=new Thread(task);
    }
```

16. Launch all the five tasks created earlier.

```
    for (int i=0; i<threads.length; i++) {
      threads[i].start();
    }
```

17. Wait for the finalization of the threads using the `join()` method.

```
    for (int i=0; i<threads.length; i++) {
      threads[i].join();
    }
```

18. Write to the console the events stored in the queue. While the size of the queue is bigger than zero, use the `poll()` method to obtain an `Event` class. If it returns `null`, put the main thread for 500 milliseconds to wait for the activation of more events.

```
    do {
      int counter=0;
      Event event;
      do {
        event=queue.poll();
        if (event!=null) counter++;
      } while (event!=null);
      System.out.printf("At %s you have read %d events\n",new
Date(),counter);
      TimeUnit.MILLISECONDS.sleep(500);
    } while (queue.size()>0);
  }

}
```

## How it works...

In this recipe, we have implemented the `Event` class. That class has a unique attribute, the activation date of the events, and implements the `Delayed` interface, so you can store `Event` objects in the `DelayedQueue` class.

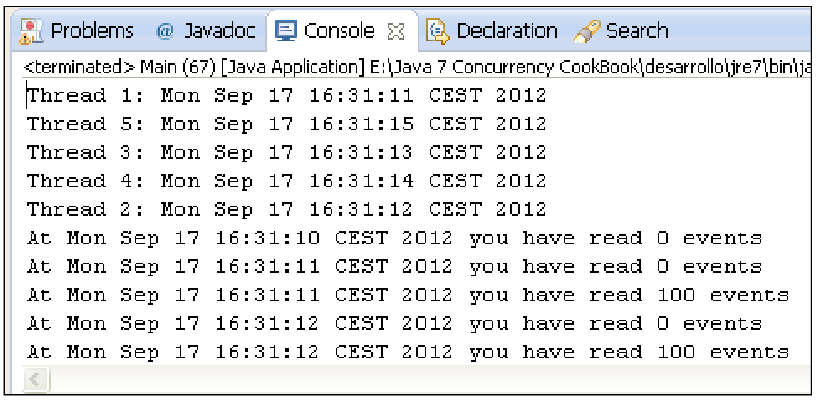
The `getDelay()` method returns the number of nanoseconds between the activation date and the actual date. Both dates are objects of the `Date` class. You have used the `getTime()` method that returns a date converted to milliseconds and then, you have converted that value to `TimeUnit` received as a parameter. The `DelayedQueue` class works in nanoseconds, but at this point, it's transparent to you.

The `compareTo()` method returns a value less than zero if the delay of the object executing the method is smaller than the delay of the object passed as a parameter, a value greater than zero if the delay of the object executing the method is bigger than the delay of the object passes as a parameter, and the value `0` if both delays are equal.

You also have implemented the `Task` class. This class has an `integer` attribute named `id`. When a `Task` object is executed, it adds a number of seconds equal to the ID of the task to the actual date and that is the activation date of the events stored by this task in the `DelayedQueue` class. Each `Task` object stores 100 events in the queue using the `add()` method.

Finally, in the `main()` method of the `Main` class, you have created five `Task` objects and executed them in the corresponding threads. When those threads finish their execution, you have written to the console all the events using the `poll()` method. That method retrieves and removes the first element of the queue. If the queue does not have any active element, the method returns the `null` value. You called the `poll()` method and if it returns an `Event` class, you increment a counter. When the `poll()` method returns the `null` value, you write the value of the counter in the console and put the thread to sleep during half a second to wait for more active events. When you have obtained the 500 events stored in the queue, the execution of the program finishes.

The following screenshot shows part of the output of an execution of the program:




You can see how the program only gets 100 events when it is activated.

> You must be very careful with the `size()` method. It returns the total number of elements in the list that includes the active and non-active elements.

## There's more...

The `DelayQueue` class has other interesting methods, which are as follows:

- `clear()`: This method removes all the elements of the queue.
- `offer(E e)`: `E` represents the class used to parameterize the `DelayQueue` class. This method inserts the element passed as a parameter in the queue.
- `peek()`: This method retrieves, but doesn't remove the first element of the queue.
- `take()`: This method retrieves and removes the first element of the queue. If there aren't any active elements in the queue, the thread that is executing the method will be blocked until the thread has some active elements.

## See also

- The *Using blocking thread-safe lists* recipe in *Chapter 6*, *Concurrent Collections*

# Using thread-safe navigable maps

An interesting data structure provided by the Java API that you can use in your concurrent programs is defined by the `ConcurrentNavigableMap` interface. The classes that implement the `ConcurrentNavigableMap` interface stores elements within two parts:

- A **key** that uniquely identifies an element
- The rest of the data that defines the element

Each part has to be implemented in different classes.

Java API also provides a class that implements that interface, which is the `ConcurrentSkipListMap` interface that implements a non-blocking list with the behavior of the `ConcurrentNavigableMap` interface. Internally, it uses a **Skip List** to store the data. A Skip List is a data structure based on parallel lists that allows us to get efficiency similar to a binary tree. With it, you can get a sorted data structure with a better access time to insert, search, or delete elements than a sorted list.

> Skip List was introduced by William Pugh in 1990.

When you insert an element in the map, it uses the key to order them, so all the elements will be ordered. The class also provides methods to obtain a submap of the map, in addition to the ones that return a concrete element.

In this recipe, you will learn how to use the `ConcurrentSkipListMap` class to implement a map of contacts.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Contact`.

```
public class Contact {
```

2. Declare two private `String` attributes named `name` and `phone`.

```
  private String name;
  private String phone;
```

3. Implement the constructor of the class to initialize its attributes.

```
  public Contact(String name, String phone) {
    this.name=name;
    this.phone=phone;
  }
```

4. Implement the methods to return the values of the `name` and `phone` attributes.

```
  public String getName() {
    return name;
  }

  public String getPhone() {
    return phone;
  }
```

5. Create a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable {
```

6. Declare a private `ConcurrentSkipListMap` attribute parameterized with the `String` and `Contact` classes named `map`.

```
  private ConcurrentSkipListMap<String, Contact> map;
```

7. Declare a private `String` attribute named `id` to store the ID of the current task.

```
  private String id;
```

8. Implement the constructor of the class to store its attributes.

```
  public Task (ConcurrentSkipListMap<String, Contact> map, String 
id) {
    this.id=id;
    this.map=map;
  }
```

9. Implement the `run()` method. It stores in the map 1,000 different contacts using the ID of the task and an incremental number to create the `Contact` objects. Use the `put()` method to store the contacts in the map.

```
@Override
  public void run() {
    for (int i=0; i<1000; i++) {
      Contact contact=new Contact(id, String.valueOf(i+1000));
      map.put(id+contact.getPhone(), contact);
    }
  }
```

10. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

11. Create a `ConcurrentSkipListMap` object parameterized with the `String` and `Conctact` classes named `map`.

```
    ConcurrentSkipListMap<String, Contact> map;
    map=new ConcurrentSkipListMap<>();
```

12. Create an array for 25 `Thread` objects to store all the `Task` objects that you're going to execute.

```
    Thread threads[]=new Thread[25];
    int counter=0;
```

13. Create and launch 25 task objects assigning a capital letter as the ID of each task.

```
    for (char i='A'; i<'Z'; i++) {
      Task task=new Task(map, String.valueOf(i));
      threads[counter]=new Thread(task);
      threads[counter].start();
      counter++;
    }
```

14. Wait for the finalization of the threads using the `join()` method.

```
    for (int i=0; i<25; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

15. Get the first entry of the map using the `firstEntry()` method. Write its data to the console.

```
    System.out.printf("Main: Size of the map: %d\n",map.size());

    Map.Entry<String, Contact> element;
    Contact contact;

    element=map.firstEntry();
    contact=element.getValue();
    System.out.printf("Main: First Entry: %s: %s\n",contact.
getName(),contact.getPhone());
```

16. Get the last entry of the map using the `lastEntry()` method. Write its data to the console.

```
    element=map.lastEntry();
    contact=element.getValue();
    System.out.printf("Main: Last Entry: %s: %s\n",contact.
getName(),contact.getPhone());
```

17. Obtain a submap of the map using the `subMap()` method. Write their data to the console.

```
    System.out.printf("Main: Submap from A1996 to B1002: \n");
    ConcurrentNavigableMap<String, Contact> submap=map.
subMap("A1996", "B1002");
    do {
      element=submap.pollFirstEntry();
      if (element!=null) {
        contact=element.getValue();
        System.out.printf("%s: %s\n",contact.getName(),contact.
getPhone());
      }
    } while (element!=null);
  }
```

## How it works...

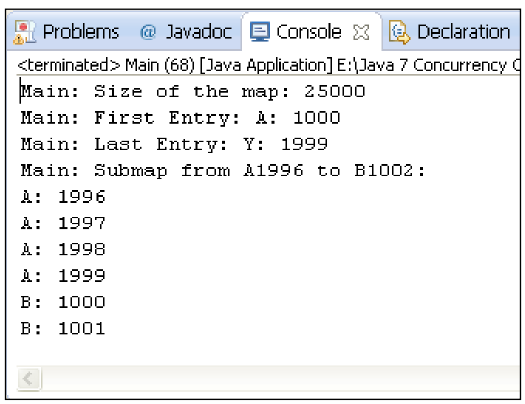
In this recipe, we have implemented a `Task` class to store `Contact` objects in the navigable map. Each contact has a name that is the ID of the task that creates it, and a phone number, that is a number between 1,000 and 2,000. We have used a concatenation of those values as the key for the contacts. Each `Task` object creates 1,000 contacts that are stored in the navigable map using the `put()` method.

> If you insert an element with a key that exists in the map, the element associated with that key will be replaced by the new element.

The `main()` method of the `Main` class creates 25 `Task` objects, using IDs as the letters between A and Z. Then, you have used some methods to obtain data from the map. The `firstEntry()` method returns a `Map.Entry` object with the first element of the map. This method doesn't remove the element from the map. That object contains the key and the element. To obtain the element, you have called the `getValue()` method. You can use the `getKey()` method to obtain the key of that element.

The `lastEntry()` method returns a `Map.Entry` object with the last element of the map and the `subMap()` method returns the `ConcurrentNavigableMap` object with part of the elements of the map, in this case, the elements which have the keys between `A1996` and `B1002`. In this case, you have used the `pollFirst()` method to process the elements of the `subMap()` method. That method returns and removes the first `Map.Entry` object of the submap.

The following screenshot shows the output of an execution of the program:




## There's more...

The `ConcurrentSkipListMap` class has other interesting methods. Following are some of them:

- `headMap(K toKey)`: `K` is the class of the key values used in the parameterization of the `ConcurrentSkipListMap` object. This method returns a submap of the first elements of the map with the elements that have a key smaller than the one passed as parameter.

- `tailMap(K fromKey)`: `K` is the class of the key values used in the parameterization of the `ConcurrentSkipListMap` object. This method returns a submap of the last elements of the map with the elements that have a key greater than the one passed as parameter.
- `putIfAbsent(K key, V Value)`: This method inserts the value specified as a parameter with the key specified as parameter if the key doesn't exist in the map.
- `pollLastEntry()`: This method returns and removes a `Map.Entry` object with the last element of the map.
- `replace(K key, V Value)`: This method replaces the value associated with the key specified as parameter if this key exists in the map.

## See also

- The *Using non-blocking thread-safe lists* recipe in *Chapter 6, Concurrent Collections*

# Generating concurrent random numbers

The Java concurrency API provides a specific class to generate pseudo-random numbers in concurrent applications. It's the `ThreadLocalRandom` class and it's new in the Java 7 Version. It works as the thread local variables. Every thread that wants to generate random numbers has a different generator, but all of them are managed from the same class, in a transparent way to the programmer. With this mechanism, you will get a better performance than using a shared `Random` object to generate the random numbers of all the threads.

In this recipe, you will learn how to use the `ThreadLocalRandom` class to generate random numbers in a concurrent application.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `TaskLocalRandom` and specify that it implements the `Runnable` interface.

```
public class TaskLocalRandom implements Runnable {
```

2. Implement the constructor of the class. Use it to initialize the random-number generator to the actual thread using the `current()` method.

```
public TaskLocalRandom() {
  ThreadLocalRandom.current();
}
```

3. Implement the `run()` method. Get the name of the thread that is executing this task and write 10 random integer numbers to the console using the `nextInt()` method.

```
@Override
public void run() {
  String name=Thread.currentThread().getName();
  for (int i=0; i<10; i++){
    System.out.printf("%s: %d\n",name,ThreadLocalRandom.
current().nextInt(10));
  }
}
```

4. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

5. Create an array for three `Thread` objects.

```
    Thread threads[]=new Thread[3];
```

6. Create and launch three `TaskLocalRandom` tasks. Store the threads in the array created earlier.

```
    for (int i=0; i<3; i++) {
      TaskLocalRandom task=new TaskLocalRandom();
      threads[i]=new Thread(task);
      threads[i].start();
    }
```

## How it works...

The key of this example is in the `TaskLocalRandom` class. In the constructor of the class, we make a call to the `current()` method of the `ThreadLocalRandom` class. This is a static method that returns the `ThreadLocalRandom` object associated with the current thread, so you can generate random numbers using that object. If the thread that makes the call does not have any object associated yet, the class creates a new one. In this case, you use this method to initialize the random generator associated with this task, so it will be created in the next call to the method.

In the `run()` method of the `TaskLocalRandom` class, make a call to the `current()` method to get the random generator associated with this thread, also you make a call to the `nextInt()` method passing number 10 as parameter. This method will return a pseudo random number between zero and 10. Each task generates 10 random numbers.

## There's more...

The `ThreadLocalRandom` class also provides methods to generate `long`, `float`, and `double` numbers, and `Boolean` values. There are methods that allow you to provide a number as a parameter to generate random numbers between zero and that number. Other methods allow you to provide two parameters to generate random numbers between those numbers.

## See also

- The *Using local thread variables* recipe in *Chapter 1*, *Thread management*

# Using atomic variables

**Atomic variables** were introduced in Java Version 5 to provide atomic operations on single variables. When you work with a normal variable, each operation that you implement in Java is transformed in several instructions that is understandable by the machine when you compile the program. For example, when you assign a value to a variable, you only use one instruction in Java, but when you compile this program, this instruction is transformed in various instructions in the JVM language. This fact can provide data inconsistency errors when you work with multiple threads that share a variable.

To avoid these problems, Java introduced the atomic variables. When a thread is doing an operation with an atomic variable, if other threads want to do an operation with the same variable, the implementation of the class includes a mechanism to check that the operation is done in one step. Basically, the operation gets the value of the variable, changes the value in a local variable, and then tries to change the old value for the new one. If the old value is still the same, it does the change. If not, the method begins the operation again. This operation is called **Compare and Set**.

Atomic variables don't use locks or other synchronization mechanisms to protect the access to their values. All their operations are based on the Compare and Set operation. It's guaranteed that several threads can work with an atomic variable at a time without generating data inconsistency errors and its performance is better than using a normal variable protected by a synchronization mechanism.

In this recipe, you will learn how to use atomic variables implementing a bank account and two different tasks, one that adds money to the account and one that subtracts money from it. You will use the `AtomicLong` class in the implementation of the example.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you are using Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Account` to simulate a bank account.

    ```
    public class Account {
    ```

2. Declare a private `AtomicLong` attribute named `balance` to store the balance of the account.

    ```
      private AtomicLong balance;
    ```

3. Implement the constructor of the class to initialize its attribute.

    ```
      public Account(){
        balance=new AtomicLong();
      }
    ```

4. Implement a method named `getBalance()` to return the value of the balance attribute.

    ```
      public long getBalance() {
        return balance.get();
      }
    ```

5. Implement a method named `setBalance()` to establish the value of the balance attribute.

    ```
      public void setBalance(long balance) {
        this.balance.set(balance);
      }
    ```

6. Implement a method named `addAmount()` to increment the value of the `balance` attribute.

    ```
      public void addAmount(long amount) {
        this.balance.getAndAdd(amount);
      }
    ```

7. Implement a method named `substractAmount()` to decrement the value of the `balance` attribute.

    ```
      public void subtractAmount(long amount) {
        this.balance.getAndAdd(-amount);
      }
    ```

8. Create a class named `Company` and specify that it implements the `Runnable` interface. This class will simulate the payments made by a company.

```
public class Company implements Runnable {
```

9. Declare a private `Account` attribute named `account`.

```
  private Account account;
```

10. Implement the constructor of the class to initialize its attribute.

```
  public Company(Account account) {
    this.account=account;
  }
```

11. Implement the `run()` method of the task. Use the `addAmount()` method of the account to make 10 increments of 1,000 in its balance.

```
@Override
  public void run() {
    for (int i=0; i<10; i++){
      account.addAmount(1000);
    }
  }
```

12. Create a class named `Bank` and specify that it implements the `Runnable` interface. This class will simulate the withdrawal of money from the account.

```
public class Bank implements Runnable {
```

13. Declare a private `Account` attribute named `account`.

```
  private Account account;
```

14. Implement the constructor of the class to initialize its attribute.

```
  public Bank(Account account) {
    this.account=account;
  }
```

15. Implement the `run()` method of the task. Use the `subtractAmount()` method of the account to make 10 decrements of 1,000 in its balance.

```
@Override
  public void run() {
    for (int i=0; i<10; i++){
      account.subtractAmount(1000);
    }
  }
```

16. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

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

17. Create an `Account` object and set its balance to `1000`.

```
    Account  account=new Account();
    account.setBalance(1000);
```

18. Create a new `Company` task and a thread to execute it.

```
    Company  company=new Company(account);
    Thread companyThread=new Thread(company);
Create a new Bank task and a thread to execute it.
    Bank bank=new Bank(account);
    Thread bankThread=new Thread(bank);
```

19. Write in the console the initial balance of the account.

```
    System.out.printf("Account : Initial Balance: %d\n",account.
getBalance());
```

20. Start the threads.

```
    companyThread.start();
    bankThread.start();
```

21. Wait for the finalization of the threads using the `join()` method and write in the console the final balance of the account.

```
    try {
      companyThread.join();
      bankThread.join();
      System.out.printf("Account : Final Balance: %d\n",account.
getBalance());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

## How it works...

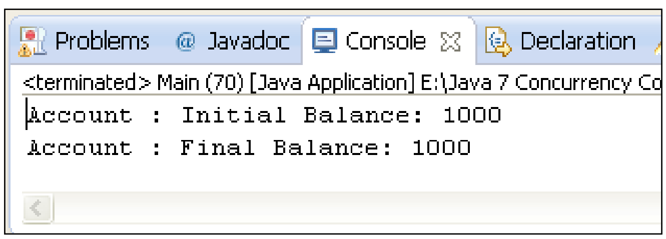
The key of this example is in the `Account` class. In that class, we declared an `AtomicLong` variable named `balance` to store the balance of the account and then we implemented the methods to work with that balance using the methods provided by the `AtomicLong` class. To implement the `getBalance()` method that returns the value of the `balance` attribute, you have used the `get()` method of the `AtomicLong` class. To implement the `setBalance()` method that establish the value of the balance attribute, you have used the `set()` method of the `AtomicLong` class. To implement the `addAmount()` method that adds an import to the balance of the account, you have used the `getAndAdd()` method of the `AtomicLong` class that returns the value and increments it by the value specified as a parameter. Finally, to implement the `subtractAmount()` method that decrements the value of the `balance` attribute, you have also used the `getAndAdd()` method.

Then, you have implemented two different tasks:

- The `Company` class simulates a company that increments the balance of the account. Each task of this class makes 10 increments of 1,000.
- The `Bank` class simulates a bank where the proprietary of the bank account takes out its money. Each task of this class makes 10 decrements of 1,000.

In the `Main` class, you have created an `Account` object with a balance of 1,000. Then, you have executed a bank task and a company task, so the final balance of the account must be the same as the initial one.

When you execute the program, you will see how the final balance is the same as the initial one. The following screenshot shows the output of an execution of this example:




## There's more...

As we mentioned in the introduction, there are other atomic classes in Java. `AtomicBoolean`, `AtomicInteger`, and `AtomicReference` are other examples of atomic classes.

## See also

- The *Synchronizing a method* recipe in *Chapter 2*, *Basic thread synchronization*

# Using atomic arrays

When you implement a concurrent application that has one or more objects shared by several threads, you have to protect the access to their attributes using a synchronization mechanism as locks or the `synchronized` keyword to avoid data inconsistency errors.

These mechanisms have the following problems:

- Deadlock: This situation occurs when a thread is blocked waiting for a lock that is locked by other threads and will never free it. This situation blocks the program, so it will never finish.
- If only one thread is accessing the shared object, it has to execute the code necessary to get and release the lock.

To provide a better performance to this situation, the **compare-and-swap operation** was developed. This operation implements the modification of the value of a variable in the following three steps:

1. You get the value of the variable, which is the old value of the variable.
2. You change the value of the variable in a temporal variable, which is the new value of the variable.
3. You substitute the old value with the new value, if the old value is equal to the actual value of the variable. The old value may be different from the actual value if another thread has changed the value of the variable.

With this mechanism, you don't need to use any synchronization mechanism, so you avoid deadlocks and you obtain a better performance.

Java implements this mechanism in the **atomic variables**. These variables provide the `compareAndSet()` method that is an implementation of the compare-and-swap operation and other methods based on it.

Java also introduced **atomic arrays** that provide atomic operations for arrays of `integer` or `long` numbers. In this recipe, you will learn how to use the `AtomicIntegerArray` class to work with atomic arrays.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Incrementer` and specify that it implements the `Runnable` interface.

   ```
   public class Incrementer implements Runnable {
   ```

2. Declare a private `AtomicIntegerArray` attribute named `vector` to store an array of `integer` numbers.

```
private AtomicIntegerArray vector;
```

3. Implement the constructor of the class to initialize its attribute.

```
public Incrementer(AtomicIntegerArray vector) {
  this.vector=vector;
}
```

4. Implement the `run()` method. Increment all the elements of the array using the `getAndIncrement()` method.

```
@Override
  public void run() {
    for (int i=0; i<vector.length(); i++){
      vector.getAndIncrement(i);
    }
  }
```

5. Create a class named `Decrementer` and specify that it implements the `Runnable` interface.

```
public class Decrementer implements Runnable {
```

6. Declare a private `AtomicIntegerArray` attribute named `vector` to store an array of `integer` numbers.

```
private AtomicIntegerArray vector;
```

7. Implement the constructor of the class to initialize its attribute.

```
public Decrementer(AtomicIntegerArray vector) {
  this.vector=vector;
}
```

8. Implement the `run()` method. Decrement all the elements of the array using the `getAndDecrement()` method.

```
@Override
  public void run() {
    for (int i=0; i<vector.length(); i++) {
      vector.getAndDecrement(i);
    }
  }
```

9. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

10. Declare a constant named `THREADS` and assign to it the value `100`. Create an `AtomicIntegerArray` object with 1,000 elements.

```
    final int THREADS=100;
    AtomicIntegerArray vector=new AtomicIntegerArray(1000);
```

11. Create an `Incrementer` task to work with the atomic array created earlier.

```
    Incrementer incrementer=new Incrementer(vector);
```

12. Create a `Decrementer` task to work with the atomic array created earlier.

```
    Decrementer decrementer=new Decrementer(vector);
```

13. Create two arrays to store 100 Thread objects.

```
    Thread threadIncrementer[]=new Thread[THREADS];
    Thread threadDecrementer[]=new Thread[THREADS];
```

14. Create and launch 100 threads to execute the `Incrementer` task and another 100 threads to execute the `Decrementer` task. Store the threads in the arrays created earlier.

```
    for (int i=0; i<THREADS; i++) {
      threadIncrementer[i]=new Thread(incrementer);
      threadDecrementer[i]=new Thread(decrementer);

      threadIncrementer[i].start();
      threadDecrementer[i].start();
    }
```

15. Wait for the finalization of the threads using the `join()` method.

```
    for (int i=0; i<100; i++) {
      try {
        threadIncrementer[i].join();
        threadDecrementer[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
```

16. Write in the console the elements of the atomic array distinct from zero. Use the `get()` method to obtain the elements of the atomic array.

```
    for (int i=0; i<vector.length(); i++) {
```

```
        if (vector.get(i)!=0) {
          System.out.println("Vector["+i+"] : "+vector.get(i));
        }
      }
```

17. Write a message in the console to indicate the finalization of the example.

```
    System.out.println("Main: End of the example");
```

## How it works...

In this example, you have implemented two different tasks to work with an `AtomicIntegerArray` object:

- `Incrementer` task: This class increments all the elements of the array using the `getAndIncrement()` method
- `Decrementer` task: This class decrements all the elements of the array using the `getAndDecrement()` method

In the `Main` class, you have created `AtomicIntegerArray` with 1,000 elements and then, you have executed 100 Incrementer and 100 decrementer tasks. At the end of those tasks, if there were no inconsistency errors, all the elements of the array must have the value `0`. If you execute the program, you will see how the program only writes to the console the final message because all the elements are zero.

## There's more...

Nowadays, Java only provides another atomic array class. It's the `AtomicLongArray` class that provides the same methods as the `IntegerAtomicArray` class.

Other interesting methods provided by these classes are:

- `get(int i)`: Returns the value of the array position specified by the parameter
- `set(int I, int newValue)`: Establishes the value of the array position specified by the parameter.

## See also

- The *Using atomic variables* recipe in *Chapter 6*, *Concurrent Collections*

# 7
# Customizing Concurrency Classes

In this chapter, we will cover:

- Customizing the `ThreadPoolExecutor` class
- Implementing a priority-based `Executor` class
- Implementing the `ThreadFactory` interface to generate custom threads
- Using our `ThreadFactory` in an `Executor` object
- Customizing tasks running in a scheduled thread pool
- Implementing the `ThreadFactory` interface to generate custom threads for the Fork/Join framework
- Customizing tasks running in the Fork/Join framework
- Implementing a custom `Lock` class
- Implementing a transfer queue based on priorities
- Implementing your own atomic object

## Introduction

Java concurrency API provides a lot of interfaces and classes to implement concurrent applications. They provide low-level mechanisms, such as the `Thread` class, the `Runnable` or `Callable` interfaces, or the `synchronized` keyword, and also high-level mechanisms, such as the Executor framework and the Fork/Join framework added in the Java 7 release. Despite this, you may find yourself developing a program where none of the java classes meet your needs.

In this case, you may need to implement your own custom-concurrent utilities based on the ones provided by Java. Basically, you can:

- Implement an interface to provide the functionality defined by that interface. For example, the `ThreadFactory` interface.
- Override some methods of a class to adapt its behavior to your needs. For example, overriding the `run()` method of the `Thread` class which, by default, does nothing useful and is supposed to be overridden to offer some functionality.

Through the recipes of this chapter, you will learn how to change the behavior of some Java concurrency API classes without the need to design a concurrency framework from scratch. You can use these recipes as an initial point to implement your own customizations.

# Customizing the ThreadPoolExecutor class

The Executor framework is a mechanism that allows you to separate the thread creation from its execution. It's based on the `Executor` and `ExecutorService` interfaces with the `ThreadPoolExecutor` class that implements both interfaces. It has an internal pool of threads and provides methods that allow you to send two kinds of tasks for their execution in the pooled threads. These tasks are:

- The `Runnable` interface to implement tasks that don't return a result
- The `Callable` interface to implement tasks that return a result

In both cases, you only send the task to the executor. The executor uses one of its pooled threads or creates a new one to execute those tasks. The executor also decides the moment in which the task is executed.

In this recipe, you will learn how to override some methods of the `ThreadPoolExecutor` class to calculate the execution time of the tasks that you execute in the executor and to write in the console statistics about the executor when it completes its execution.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow the steps described below to implement the example:

1. Create a class named `MyExecutor` that extends the `ThreadPoolExecutor` class.

```
public class MyExecutor extends ThreadPoolExecutor {
```

2. Declare a private `ConcurrentHashMap` attribute parameterized with the `String` and `Date` classes named `startTimes`.

```
  private ConcurrentHashMap<String, Date> startTimes;
```

3. Implement the constructor for the class. Call a constructor of the parent class using the `super` keyword and initialize the `startTime` attribute.

```
  public MyExecutor(int corePoolSize, int maximumPoolSize,
      long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable>
workQueue) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue);
    startTimes=new ConcurrentHashMap<>();
  }
```

4. Override the `shutdown()` method. Write in the console information about the executed tasks, the running tasks, and the pending tasks. Then, call the `shutdown()` method of the parent class using the `super` keyword.

```
  @Override
  public void shutdown() {
    System.out.printf("MyExecutor: Going to shutdown.\n");
    System.out.printf("MyExecutor: Executed tasks:
%d\n",getCompletedTaskCount());
    System.out.printf("MyExecutor: Running tasks:
%d\n",getActiveCount());
    System.out.printf("MyExecutor: Pending tasks:
%d\n",getQueue().size());
    super.shutdown();
  }
```

5. Override the `shutdownNow()` method. Write in the console information about the executed tasks, the running tasks, and the pending tasks. Then, call the `shutdownNow()` method of the parent class using the `super` keyword.

```
  @Override
  public List<Runnable> shutdownNow() {
    System.out.printf("MyExecutor: Going to immediately
shutdown.\n");
    System.out.printf("MyExecutor: Executed tasks:
%d\n",getCompletedTaskCount());
    System.out.printf("MyExecutor: Running tasks:
%d\n",getActiveCount());
    System.out.printf("MyExecutor: Pending tasks:
%d\n",getQueue().size());
    return super.shutdownNow();
  }
```

6. Override the `beforeExecute()` method. Write a message in the console with the name of the thread that is going to execute the task and the hash code of the task. Store the start date in `HashMap` using the hash code of the task as the key.

```
  @Override
  protected void beforeExecute(Thread t, Runnable r) {
    System.out.printf("MyExecutor: A task is beginning: %s :
%s\n",t.getName(),r.hashCode());
    startTimes.put(String.valueOf(r.hashCode()), new Date());
  }
```

7. Override the `afterExecute()` method. Write a message in the console with the result of the task and calculate the running time of the task subtracting the start date of the task stored in `HashMap` of the current date.

```
  @Override
  protected void afterExecute(Runnable r, Throwable t) {
    Future<?> result=(Future<?>)r;
    try {
      System.out.printf("*********************************\n");
      System.out.printf("MyExecutor: A task is finishing.\n");
      System.out.printf("MyExecutor: Result: %s\n",result.get());
      Date startDate=startTimes.remove(String.valueOf(r.
hashCode()));
      Date finishDate=new Date();
      long diff=finishDate.getTime()-startDate.getTime();
      System.out.printf("MyExecutor: Duration: %d\n",diff);
      System.out.printf("*********************************\n");
    } catch (InterruptedException  | ExecutionException e) {
      e.printStackTrace();
    }
  }
}
```

8. Create a class named `SleepTwoSecondsTask` that implements the `Callable` interface parameterized with the `String` class. Implement the `call()` method. Put the current thread to sleep for 2 seconds and return the current date converted to a `String` type.

```
public class SleepTwoSecondsTask implements Callable<String> {

  public String call() throws Exception {
    TimeUnit.SECONDS.sleep(2);
    return new Date().toString();
  }

}
```

9. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

10. Create a `MyExecutor` object named `myExecutor`.

```
    MyExecutor myExecutor=new MyExecutor(2, 4, 1000, TimeUnit.
MILLISECONDS, new LinkedBlockingDeque<Runnable>());
```

11. Create a list of `Future` objects parameterized with the `String` class to store the resultant objects of the tasks you're going to send to the executor.

```
    List<Future<String>> results=new ArrayList<>();¡;
```

12. Submit 10 `Task` objects.

```
    for (int i=0; i<10; i++) {
      SleepTwoSecondsTask task=new SleepTwoSecondsTask();
      Future<String> result=myExecutor.submit(task);
      results.add(result);
    }
```

13. Get the result of the execution of the first five tasks using the `get()` method. Write them in the console.

```
    for (int i=0; i<5; i++){
      try {
        String result=results.get(i).get();
        System.out.printf("Main: Result for Task %d :
%s\n",i,result);
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
```

14. Finish the execution of the executor using the `shutdown()` method.

```
    myExecutor.shutdown();
```

15. Get the result of the execution of the last five tasks using the `get()` method. Write them in the console.

```
    for (int i=5; i<10; i++){
      try {
        String result=results.get(i).get();
        System.out.printf("Main: Result for Task %d :
%s\n",i,result);
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
```

16. Wait for the completion of the executor using the `awaitTermination()` method.

```
try {
  myExecutor.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

17. Write a message indicating the end of the execution of the program.

```
System.out.printf("Main: End of the program.\n");
```

## How it works...

In this recipe, we have implemented our custom executor extending the `ThreadPoolExecutor` class and overriding four of its methods. The `beforeExecute()` and `afterExecute()` methods were used to calculate the execution time of a task. The `beforeExecute()` method is executed before the execution of a task. In this case, we have used `HashMap` to store in it the start date of the task. The `afterExecute()` method is executed after the execution of a task. You get `startTime` of the task that has finished from `HashMap` and then, calculate the difference between the actual date and that date to get the execution time of the task. You have also overridden the `shutdown()` and `shutdownNow()` methods to write statistics about the tasks executed in the executor to the console:

- The executed tasks, using the `getCompletedTaskCount()` method
- The tasks that are running at this time, using the `getActiveCount()` method

The pending tasks, using the `size()` method of the blocking queue where the executor stores the pending tasks. The `SleepTwoSecondsTask` class that implements the `Callable` interface puts its execution thread to sleep for 2 seconds and the `Main` class, where you send 10 tasks to your executor that uses it and the other classes to demo their features.

Execute the program and you will see how the program shows the time span of each task that is running and the statistics of the executor upon calling the `shutdown()` method.

## See also

- The *Creating a Thread executor* recipe in *Chapter 4*, *Thread Executors*
- The *Using our ThreadFactory in an Executor* object recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Implementing a priority-based Executor class

In the first versions of the Java concurrency API, you had to create and run all the threads of your application. In Java Version 5, with the appearance of the Executor framework, a new mechanism was introduced for the execution of concurrency tasks.

With the Executor framework, you only have to implement your tasks and send them to the executor. The executor is responsible for the creation and execution of the threads that execute your tasks.

Internally, an executor uses a blocking queue to store pending tasks. These are stored in the order of their arrival to the executor. One possible alternative is the use of a priority queue to store new tasks. In this way, if a new task with high priority arrives to the executor, it will be executed before other threads that have already been waiting for a thread to execute, but have lower priority.

In this recipe, you will learn how to implement an executor that will use a priority queue to store the tasks you send for execution.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyPriorityTask` that implements the `Runnable` and `Comparable` interfaces parameterized with the `MyPriorityTask` class interface.

   ```
   public class MyPriorityTask implements Runnable,
   Comparable<MyPriorityTask> {
   ```

2. Declare a private `int` attribute named `priority`.

   ```
     private int priority;
   ```

3. Declare a private `String` attribute named `name`.

   ```
     private String name;
   ```

4. Implement the constructor of the class to initialize its attributes.

```
public MyPriorityTask(String name, int priority) {
  this.name=name;
  this.priority=priority;
}
```

5. Implement a method to return the value of the priority attribute.

```
public int getPriority(){
  return priority;
}
```

6. Implement the `compareTo()` method declared in the `Comparable` interface. It receives a `MyPriorityTask` object as a parameter and compares the priorities of the two objects, the current one and the parameter. You let the tasks with higher priority execute before the tasks with lower priority.

```
@Override
public int compareTo(MyPriorityTask o) {
  if (this.getPriority() < o.getPriority()) {
    return 1;
  }
  if (this.getPriority() > o.getPriority()) {
    return -1;
  }
  return 0;
}
```

7. Implement the `run()` method. Put the current thread to sleep for 2 seconds.

```
  @Override
public void run() {
  System.out.printf("MyPriorityTask: %s Priority :
%d\n",name,priority);
  try {
    TimeUnit.SECONDS.sleep(2);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

8. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) {
```

9. Create a `ThreadPoolExecutor` object named `executor`. Use `PriorityBlockingQueue` parameterized with the `Runnable` interface as the queue that this executor will use to store its pending tasks.

```
    ThreadPoolExecutor executor=new ThreadPoolExecutor(2,2,1,TimeU
nit.SECONDS,new PriorityBlockingQueue<Runnable>());
```

10. Send four tasks to the executor using the counter of the loop as priority of the tasks. Use the `execute()` method to send the tasks to the executor.

```
    for (int i=0; i<4; i++){
      MyPriorityTask task=new MyPriorityTask ("Task "+i,i);
      executor.execute(task);
    }
```

11. Put the current thread to sleep for 1 second.

```
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

12. Send four additional tasks to the executor using the counter of the loop as priority of the tasks. Use the `execute()` method to send the tasks to the executor.

```
    for (int i=4; i<8; i++) {
      MyPriorityTask task=new MyPriorityTask ("Task "+i,i);
      executor.execute(task);
    }
```

13. Shut down the executor using the `shutdown()` method.

```
    executor.shutdown();
```

14. Wait for the finalization of the executor using the `awaitTermination()` method.

```
    try {
      executor.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

15. Write a message in the console indicating the finalization of the program.

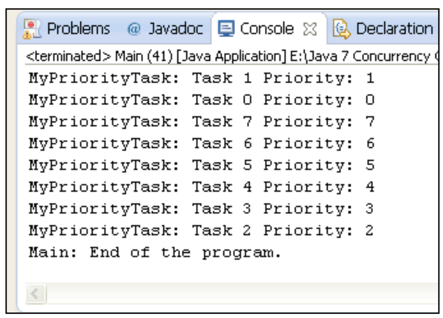
```
    System.out.printf("Main: End of the program.\n");
```

## How it works...

To convert an executor to a priority-based one is simple. You only have to pass a `PriorityBlockingQueue` object parameterized with the `Runnable` interface as a parameter. But with the executor, you should know that all the objects stored in a priority queue have to implement the `Comparable` interface.

You have implemented the `MyPriorityTask` class that implements the `Runnable` interface, to be a task, and the `Comparable` interface, to be stored in the priority queue. This class has a `Priority` attribute that is used to store the priority of the tasks. If a task has a higher value for this attribute, it will be executed earlier. The `compareTo()` method determines the order of the tasks in the priority queue. In the `Main` class, you sent eight tasks to the executor with different priorities. The first tasks you sent to the executor are the first tasks that are executed. As the executor is idle waiting for tasks to be executed, and as the first tasks arrive to the executor, it executes them immediately. You have created the executor with two execution threads, so the first two tasks will be the first ones that are executed. Then, the rest of the tasks are executed based on their priority.

The following screenshot shows one execution of this example:




## There's more...

You can configure `Executor` to use any implementation of the `BlockingQueue` interface. One interesting implementation is `DelayQueue`. This class is used to store elements with a delayed activation. It provides methods that only return the active objects. You can use this class to implement your own version of the `ScheduledThreadPoolExecutor` class.

## See also

- The *Creating a Thread Executor* recipe in *Chapter 4*, *Thread Executors*
- The *Customizing the ThreadPoolExecutor* class recipe in *Chapter 7*, *Customizing Concurrency Classes*

- The *Using blocking thread-safe lists ordered by priority* recipe in *Chapter 6, Concurrent Collections*

# Implementing the ThreadFactory interface to generate custom threads

The **factory pattern** is a widely used design pattern in the object-oriented programming world. It is a creational pattern and its objective is to develop a class whose mission will be creating objects of one or several classes. Then, when we want to create an object of one of those classes, we use the factory instead of using the new operator.

- With this factory, we centralize the creation of objects gaining an advantage of easily changing the class of objects created or the way we create these objects that are easily limiting the creation of objects for limited resources. For example, we can only have *N* objects of a type that is easily generating statistical data about the creation of objects.

Java provides the `ThreadFactory` interface to implement a `Thread` object factory. Some advanced utilities of the Java concurrency API, as the Executor framework or the Fork/Join framework, use thread factories to create threads.

Another example of the factory pattern in the Java Concurrency API is the `Executors` class. It provides a lot of methods to create different kinds of `Executor` objects.

In this recipe, you will extend the `Thread` class by adding new functionalities and you will implement a thread factory class to generate threads of that new class.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or another IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyThread` that extends the `Thread` class.

```
public class MyThread extends Thread {
```

2. Declare three private `Date` attributes named `creationDate`, `startDate`, and `finishDate`.

```
  private Date creationDate;
  private Date startDate;
  private Date finishDate;
```

3. Implement a constructor of the class. It receives the name and the `Runnable` object to execute as parameters. Store the creation date of the thread.

```
public MyThread(Runnable target, String name ){
  super(target,name);
  setCreationDate();
}
```

4. Implement the `run()` method. Store the start date of the thread, call the `run()` method of the parent class, and store the finish date of the execution.

```
@Override
public void run() {
  setStartDate();
  super.run();
  setFinishDate();
}
```

5. Implement a method to establish the value of the `creationDate` attribute.

```
public void setCreationDate() {
  creationDate=new Date();
}
```

6. Implement a method to establish the value of the `startDate` attribute.

```
public void setStartDate() {
  startDate=new Date();
}
```

7. Implement a method to establish the value of the `finishDate` attribute.

```
public void setFinishDate() {
  finishDate=new Date();
}
```

8. Implement a method named `getExecutionTime()` that calculates the execution time of the thread as the difference between the start date and the finish date.

```
public long getExecutionTime() {
  return finishDate.getTime()-startDate.getTime();
}
```

9. Override the `toString()` method to return the creation date and the execution time of the thread.

```
@Override
public String toString(){
  StringBuilder buffer=new StringBuilder();
  buffer.append(getName());
  buffer.append(": ");
```

```
    buffer.append(" Creation Date: ");
    buffer.append(creationDate);
    buffer.append(" : Running time: ");
    buffer.append(getExecutionTime());
    buffer.append(" Milliseconds.");
    return buffer.toString();
  }
```

10. Create a class named `MyThreadFactory` that implements the `ThreadFactory` interface.

```
public class MyThreadFactory implements ThreadFactory {
```

11. Declare a private `int` attribute named `counter`.

```
  private int counter;
```

12. Declare a private `String` attribute named `prefix`.

```
  private String prefix;
```

13. Implement the constructor of the class to initialize its attributes.

```
  public MyThreadFactory (String prefix) {
    this.prefix=prefix;
    counter=1;
  }
```

14. Implement the `newThread()` method. Create a `MyThread` object and increment the `counter` attribute.

```
  @Override
  public Thread newThread(Runnable r) {
    MyThread myThread=new MyThread(r,prefix+"-"+counter);
    counter++;
    return myThread;
  }
```

15. Create a class named `MyTask` that implements the `Runnable` interface. Implement the `run()` method. Put the current thread to sleep for 2 seconds.

```
public class MyTask implements Runnable {
  @Override
  public void run() {
    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  }
```

16. Implement the main class of the example by creating a class named `Main` with a `main()` method.

    ```
    public class Main {
      public static void main(String[] args) throws Exception {
    ```

17. Create a `MyThreadFactory` object.

    ```
        MyThreadFactory myFactory=new MyThreadFactory("MyThreadFactory
    ");
    ```

18. Create a `Task` object.

    ```
        MyTask task=new MyTask();
    ```

19. Create a `MyThread` object to execute the task using the `newThread()` method of the factory.

    ```
        Thread thread=myFactory.newThread(task);
    ```

20. Start the thread and wait for its finalization.

    ```
        thread.start();
        thread.join();
    ```

21. Write information about the thread using the `toString()` method.

    ```
        System.out.printf("Main: Thread information.\n");
        System.out.printf("%s\n",thread);
        System.out.printf("Main: End of the example.\n");
    ```

## How it works...

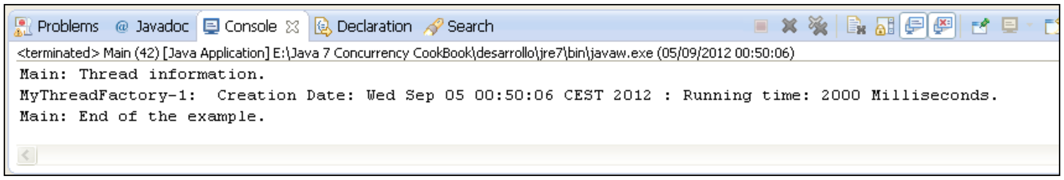
In this recipe, you have implemented a custom `MyThread` class extending the `Thread` class. The class has three attributes to store the creation date, the start date of its execution, and the end date of its execution. Using the start date and the end date attributes, you have implemented the `getExecutionTime()` method that returns the time that the thread has been executing its task. Finally, you have overridden the `toString()` method to generate information about a thread.

Once you had your own thread class, you have implemented a factory to create objects of that class implementing the `ThreadFactory` interface. It's not mandatory to make use of the interface if you're going to use your factory as an independent object, but if you want to use this factory with other classes of the Java concurrency API, you must construct your factory by implementing that interface. The `ThreadFactory` interface only has one method, the `newThread()` method that receives a `Runnable` object as a parameter and returns a `Thread` object to execute that `Runnable` object. In your case, you return a `MyThread` object.

To check these two classes, you have implemented the `MyTask` class that implements the `Runnable` object. This is the task to be executed in threads managed by the `MyThread` object. A `MyTask` instance puts its execution thread to sleep for 2 seconds.

In the main method of the example, you have created a `MyThread` object using a `MyThreadFactory` factory to execute a `Task` object. Execute the program and you will see a message with the start date and the execution time of the thread executed.

The following screenshot shows the output generated by this example:




## There's more...

The Java Concurrency API provides the `Executors` class to generate thread executors, usually objects of the `ThreadPoolExecutor` class. You can also use this class to obtain the most basic implementation of the `ThreadFactory` interface using the `defaultThreadFactory()` method. The factory generated by this method generates basic `Thread` objects belonging to all of them to the same `ThreadGroup` object.

You can use the `ThreadFactory` interface in your program for any purposes, not necessarily related to the Executor framework.

# Using our ThreadFactory in an Executor object

In the previous recipe, *Implementing the ThreadFactory interface to generate custom threads*, we introduced the factory pattern and provided an example of how to implement a factory of threads implementing the `ThreadFactory` interface.

The Executor framework is a mechanism that allows you the separation of the thread creation and its execution. It's based on the `Executor` and `ExecutorService` interfaces and in the `ThreadPoolExecutor` class that implements both interfaces. It has an internal pool of threads and provides methods that allow you to send two kinds of tasks for their execution in the pooled threads. These two kinds of tasks are:

- Classes that implement the `Runnable` interface, to implement tasks that don't return a result
- Classes that implement the `Callable` interface, to implement tasks that return a result

Internally, an Executor framework uses a `ThreadFactory` interface to create the threads that it uses to generate the new threads. In this recipe, you will learn how to implement your own thread class, a thread factory to create threads of that class, and how to use that factory in an executor, so the executor will execute your threads.

## Getting ready...

Read the previous recipe, *Implementing a ThreadFactory interface to generate custom threads*, and implement its example.

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or another IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Copy into the project the classes `MyThread`, `MyThreadFactory`, and `MyTask` implemented in the recipe *Implementing a ThreadFactory interface to generate custom threads*, so you are going to use them in this example.
2. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {
  public static void main(String[] args) throws Exception {
```

3. Create a new `MyThreadFactory` object named `threadFactory`.

```
    MyThreadFactory threadFactory=new MyThreadFactory
    ("MyThreadFactory");
```

4. Create a new `Executor` object using the `newCachedThreadPool()` method of the `Executors` class. Pass the factory object created earlier as a parameter. The new `Executor` object will use that factory to create the necessary threads, so it will execute `MyThread` threads.

```
    ExecutorService executor=Executors.newCachedThreadPool
    (threadFactory);
```

5. Create a new `Task` object and send it to the executor using the `submit()` method.

```
    MyTask task=new MyTask();
    executor.submit(task);
```

6. Shut down the executor using the `shutdown()` method.

```
    executor.shutdown();
```

7. Wait for the finalization of the executor using the `awaitTermination()` method.

```
executor.awaitTermination(1, TimeUnit.DAYS);
```

8. Write a message to indicate the end of the program.

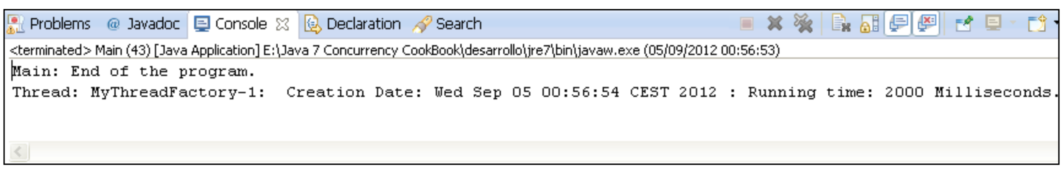
```
System.out.printf("Main: End of the program.\n");
```

## How it works...

In the *How it works...* section of the previous recipe, *Implementing a ThreadFactory interface to generate custom threads*, you can read a detailed explanation of how the `MyThread`, `MyThreadFactory`, and `MyTask` classes works.

In the `main()` method of the example, you have created an `Executor` object using the `newCachedThreadPool()` method of the `Executors` class. You have passed the factory object created earlier as a parameter, so the `Executor` object created will use that factory to create the threads it needs and it will execute threads of the `MyThread` class.

Execute the program and you will see a message with information about the thread's start date and its execution time. The following screenshot shows the output generated by this example:




## See also

- The *Implementing a ThreadFactory interface to generate custom threads* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Customizing tasks running in a scheduled thread pool

The **scheduled thread pool** is an extension of the basic thread pool of the Executor framework that allows you to schedule the execution of tasks to be executed after a period of time. It's implemented by the `ScheduledThreadPoolExecutor` class and it permits the execution of the following two kinds of tasks:

- **Delayed tasks**: These kinds of tasks are executed only once after a period of time
- **Periodic tasks**: These kinds of tasks are executed after a delay and then periodically every so often

Delayed tasks can execute both, the `Callable` and `Runnable` objects, but the periodic tasks can only execute `Runnable` objects. All the tasks executed by a scheduled pool are an implementation of the `RunnableScheduledFuture` interface. In this recipe, you will learn how to implement your own implementation of the `RunnableScheduledFuture` interface to execute delayed and periodic tasks.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps described below to implement the example:

1. Create a class named `MyScheduledTask` parameterized with a generic type named `V`. It extends the `FutureTask` class and implements the `RunnableScheduledFuture` interface.

   ```
   public class MyScheduledTask<V> extends FutureTask<V> implements
   RunnableScheduledFuture<V> {
   ```

2. Declare a private `RunnableScheduledFuture` attribute named `task`.

   ```
     private RunnableScheduledFuture<V> task;
   ```

3. Declare a private `ScheduledThreadPoolExecutor` named `executor`.

   ```
     private ScheduledThreadPoolExecutor executor;
   ```

4. Declare a private `long` attribute named `period`.

   ```
     private long period;
   ```

5. Declare a private `long` attribute named `startDate`.

   ```
     private long startDate;
   ```

6. Implement a constructor of the class. It receives the `Runnable` object that is going to be executed by a task, the result that will be returned by this task, the `RunnableScheduledFuture` task that will be used to create the `MyScheduledTask` object, and the `ScheduledThreadPoolExecutor` object that is going to execute the task. Call the constructor of its parent class and store the task and `executor` attributes.

   ```
     public MyScheduledTask(Runnable runnable, V result,
   RunnableScheduledFuture<V> task, ScheduledThreadPoolExecutor
   executor) {
       super(runnable, result);
       this.task=task;
   ```

```
    this.executor=executor;
  }
```

7. Implement the `getDelay()` method. If the task is a periodic task and the `startDate` attribute has a value different from zero, calculate the returned value as the difference between the `startDate` attribute and the actual date. Otherwise, return the delay of the original task stored in the `task` attribute. Don't forget that you have to return the result in the time unit passed as a parameter.

```
  @Override
  public long getDelay(TimeUnit unit) {
    if (!isPeriodic()) {
      return task.getDelay(unit);
    } else {
      if (startDate==0){
        return task.getDelay(unit);
      } else {
        Date now=new Date();
        long delay=startDate-now.getTime();
        return unit.convert(delay, TimeUnit.MILLISECONDS);
      }
    }
  }
```

8. Implement the `compareTo()` method. Call the `compareTo()` method of the original task.

```
  @Override
  public int compareTo(Delayed o) {
    return task.compareTo(o);
  }
```

9. Implement the `isPeriodic()` method. Call the `isPeriodic()` method of the original task.

```
  @Override
  public boolean isPeriodic() {
    return task.isPeriodic();
  }
```

10. Implement the `run()` method. If it's a periodic task, you have to update its `startDate` attribute with the start date of the next execution of the task. Calculate it as the sum of the actual date and the period. Then, add the task again to the queue of the `ScheduledThreadPoolExecutor` object.

```
  @Override
  public void run() {
    if (isPeriodic() && (!executor.isShutdown())) {
      Date now=new Date();
```

```
      startDate=now.getTime()+period;
      executor.getQueue().add(this);
    }
```

11. Print a message with the actual date to the console, execute the task calling the `runAndReset()` method, and then print another message with the actual date to the console.

```
    System.out.printf("Pre-MyScheduledTask: %s\n",new Date());
    System.out.printf("MyScheduledTask: Is Periodic:
%s\n",isPeriodic());
    super.runAndReset();
    System.out.printf("Post-MyScheduledTask: %s\n",new Date());
  }
```

12. Implement the `setPeriod()` method to establish the period of this task.

```
  public void setPeriod(long period) {
    this.period=period;
  }
```

13. Create a class named `MyScheduledThreadPoolExecutor` to implement a `ScheduledThreadPoolExecutor` object that executes `MyScheduledTask` tasks. Specify that this class extends the `ScheduledThreadPoolExecutor` class.

```
public class MyScheduledThreadPoolExecutor extends
ScheduledThreadPoolExecutor {
```

14. Implement a constructor of the class which merely calls the constructor of its parent class.

```
  public MyScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize);
  }
```

15. Implement the `decorateTask()` method. It receives as a parameter the `Runnable` object that is going to be executed and the `RunnableScheduledFuture` task that will execute that `Runnable` object. Create and return a `MyScheduledTask` task using those objects to construct them.

```
  @Override
  protected <V> RunnableScheduledFuture<V> decorateTask(Runnable
runnable,
      RunnableScheduledFuture<V> task) {
    MyScheduledTask<V> myTask=new MyScheduledTask<V>(runnable,
null, task,this);
    return myTask;
  }
```

16. Override the `scheduledAtFixedRate()` method. Call the method of its parent class, convert the returned object into a `MyScheduledTask` object, and establish the period of that task using the `setPeriod()` method.

```
  @Override
  public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
long initialDelay, long period, TimeUnit unit) {
    ScheduledFuture<?> task= super.scheduleAtFixedRate(command,
initialDelay, period, unit);
    MyScheduledTask<?> myTask=(MyScheduledTask<?>)task;
 myTask.setPeriod(TimeUnit.MILLISECONDS.convert(period,unit));
    return task;
 }
```

17. Create a class named `Task` that implements the `Runnable` interface.

```
public class Task implements Runnable {
```

18. Implement the `run()` method. Print a message at the start of the task, put the current thread to sleep 2 seconds, and print another message at the end of the task.

```
  @Override
  public void run() {
    System.out.printf("Task: Begin.\n");
    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("Task: End.\n");
  }
```

19. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

20. Create a `MyScheduledThreadPoolExecutor` object named `executor`. Use `2` as a parameter to have two threads in the pool.

```
    MyScheduledThreadPoolExecutor executor=new
MyScheduledThreadPoolExecutor(2);
```

21. Create a `Task` object named `task`. Write the actual date in the console.

```
    Task task=new Task();
    System.out.printf("Main: %s\n",new Date());
```

22. Send a delayed task to the executor using the `schedule()` method. The task will be executed after a 1 second delay.

    ```
    executor.schedule(task, 1, TimeUnit.SECONDS);
    ```

23. Put the main thread to sleep for 3 seconds.

    ```
    TimeUnit.SECONDS.sleep(3);
    ```

24. Create another `Task` object. Print the actual date in the console again.

    ```
    task=new Task();
        System.out.printf("Main: %s\n",new Date());
    ```

25. Send a periodic task to the executor using the `scheduleAtFixedRate()` method. The task will be executed after a 1 second delay and then will be executed every 3 seconds.

    ```
    executor.scheduleAtFixedRate(task, 1, 3, TimeUnit.SECONDS);
    ```

26. Put the main thread to sleep for 10 seconds.

    ```
    TimeUnit.SECONDS.sleep(10);
    ```

27. Shut down the executor using the `shutdown()` method. Wait for the finalization of the executor using the `awaitTermination()` method.

    ```
    executor.shutdown();
    executor.awaitTermination(1, TimeUnit.DAYS);
    ```

28. Write a message in the console indicating the end of the program.

    ```
    System.out.printf("Main: End of the program.\n");
    ```

## How it works...

In this recipe, you have implemented the `MyScheduledTask` class to implement a custom task that can execute on a `ScheduledThreadPoolExecutor` executor. This class extends the `FutureTask` class and implements the `RunnableScheduledFuture` interface. It implements the `RunnableScheduledFuture` interface, because all the tasks executed in a scheduled executor must implement that interface and extend the `FutureTask` class, because this class provides valid implementations of the methods declared in the `RunnableScheduledFuture` interface. All the interfaces and classes mentioned earlier are parameterized classes, with the type of data that will be returned by the tasks.

To use a `MyScheduledTask` task in a scheduled executor, you have overridden the `decorateTask()` method in the `MyScheduledThreadPoolExecutor` class. This class extends the `ScheduledThreadPoolExecutor` executor and that method provides a mechanism to convert the default scheduled tasks implemented by the `ScheduledThreadPoolExecutor` executor to `MyScheduledTask` tasks. So, when you implement your own version of scheduled tasks, you have to implement your own version of a scheduled executor.

- The `decorateTask()` method simply creates a new `MyScheduledTask` object with the parameter; a `Runnable` object that is going to be executed in the task. A resultant object that is going to be returned by that task. In this case, the task won't return a result, so you used the `null` value. An original task is used to execute the `Runnable` object. This is the task that the new object is going to replace in the pool; an executor that is going to execute the task. In this case, you use the `this` keyword to reference the executor that is creating the task.

The `MyScheduledTask` class can execute delayed and periodic tasks. You have implemented two methods with all the necessary logic to execute both kinds of tasks. They are the `getDelay()` and the `run()` methods.

The `getDelay()` method is called by the scheduled executor to know if it has to execute a task. The behavior of this method changes in delayed and periodic tasks. As we mentioned earlier, the constructor of the `MyScheduledClass` class receives the original `ScheduledRunnableFuture` object that was going to execute the `Runnable` object and stores it as an attribute of the class to have access to its methods and its data. When we are going to execute a delayed task, the `getDelay()` method returns the delay of the original task, but in the case of the periodic task, the `getDelay()` method returns the difference between the `startDate` attribute and the actual date.

The `run()` method is the one that executes the task. One particularity of the periodic tasks is that you have to put the next execution of the task in the queue of the executor as a new task if you want the task to be executed again. So, if you're executing a periodic task, you establish the `startDate` attribute value adding to the actual date and the period of execution of the task and store the task again in the queue of the executor. The `startDate` attribute stores the date when the next execution of the task will begin. Then, you execute the task using the `runAndReset()` method provided by the `FutureTask` class. In the case of the delayed tasks, you don't have to put them in the queue of the executor, because they only execute once.

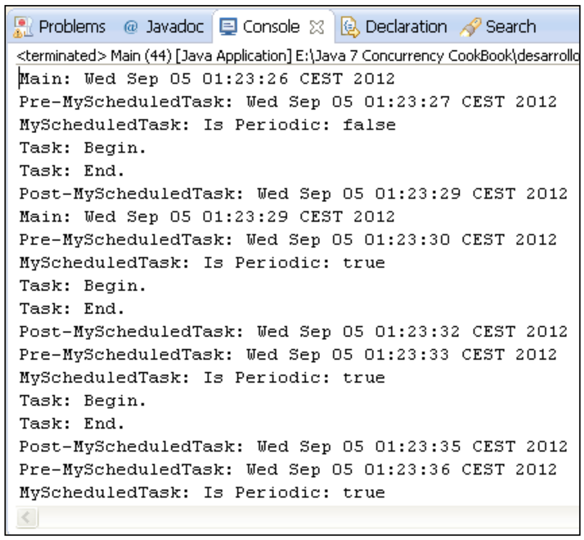
> You have also to take into account if the executor has been shutdown. In that case, you don't have to store again the periodic tasks into the queue of the executor.

Finally, you have overridden the `scheduleAtFixedRate()` method in the `MyScheduledThreadPoolExecutor` class. We mentioned earlier that, for periodic tasks, you establish the value of the `startDate` attribute using the period of the task, but you haven't initialized that period yet. You have to override this method that receives that period as a parameter, to pass it to the `MyScheduledTask` class so it can use it.

The example is complete with the `Task` class that implements the `Runnable` interface and is the task executed in the scheduled executor. The main class of the example creates a `MyScheduledThreadPoolExecutor` executor and sends the following two tasks to them:

- A delayed task, to be executed 1 second after the actual date
- A periodic task, to be executed for the first time 1 second after the actual date and then every 3 seconds

The following screenshot shows part of the execution of this example. You can check as the two kinds of tasks are executed properly:




## There's more...

The `ScheduledThreadPoolExecutor` class provides another version of the `decorateTask()` method that receives a `Callable` object as a parameter instead of a `Runnable` object.

## See also

- The *Running a task in an executor after a delay* recipe in *Chapter 4*, *Thread Executors*
- The *Running a task in an executor periodically* recipe *Chapter 4*, *Thread Executors*

# Implementing the ThreadFactory interface to generate custom threads for the Fork/Join framework

One of the most interesting features of Java 7 is the Fork/Join framework. It's an implementation of the `Executor` and `ExecutorService` interfaces that allow you the execution of the `Callable` and `Runnable` tasks without managing the threads that execute them.

This executor is oriented to execute tasks that can be divided into smaller parts. Its main components are as follows:

- A special kind of task, implemented by the `ForkJoinTask` class.
- Two operations for dividing a task into subtasks (the `fork` operation) and to wait for the finalization of those subtasks (the `join` operation).
- An algorithm, denominating the work-stealing algorithm, that optimizes the use of the threads of the pool. When a task is waiting for its subtasks, the thread that was executing it is used to execute another thread.

The main class of the Fork/Join framework is the `ForkJoinPool` class. Internally, it has the following two elements:

- A queue of tasks that are waiting to be executed
- A pool of threads that execute the tasks

In this recipe, you will learn how to implement a customized worker thread to be used in a `ForkJoinPool` class and how to use it using a factory.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyWorkerThread` that extends the `ForkJoinWorkerThread` class.

```
public class MyWorkerThread extends ForkJoinWorkerThread {
```

2. Declare and create a private `ThreadLocal` attribute parameterized with the `Integer` class named `taskCounter`.

```
  private static ThreadLocal<Integer> taskCounter=new
ThreadLocal<Integer>();
```

3. Implement  a constructor of the class.

```
  protected MyWorkerThread(ForkJoinPool pool) {
    super(pool);
  }
```

4. Override the `onStart()` method. Call the method on its parent class, print a message to the console, and set the value of the `taskCounter` attribute for this thread to zero.

```
  @Override
  protected void onStart() {
    super.onStart();
    System.out.printf("MyWorkerThread %d: Initializing task
counter.\n",getId());
    taskCounter.set(0);
  }
```

5. Override the `onTermination()` method. Write the value of the `taskCounter` attribute for this thread in the console.

```
  @Override
  protected void onTermination(Throwable exception) {
    System.out.printf("MyWorkerThread %d:
%d\n",getId(),taskCounter.get());
    super.onTermination(exception);
  }
```

6. Implement the `addTask()` method. Increment the value of the `taskCounter` attribute.

```
  public void addTask(){
    int counter=taskCounter.get().intValue();
    counter++;
    taskCounter.set(counter);
  }
```

7. Create a class named `MyWorkerThreadFactory` that implements the `ForkJoinWorkerThreadFactory` interface. Implement the `newThread()` method. Create and return a `MyWorkerThread` object.

```
public class MyWorkerThreadFactory implements
ForkJoinWorkerThreadFactory {

  @Override
  public ForkJoinWorkerThread newThread(ForkJoinPool pool) {
```

```
    return new MyWorkerThread(pool);
  }

}
```

8. Create a class named `MyRecursiveTask` that extends the `RecursiveTask` class parameterized with the `Integer` class.

```
public class MyRecursiveTask extends RecursiveTask<Integer> {
```

9. Declare a private `int` array named `array`.

```
  private int array[];
```

10. Declare two private `int` attributes named `start` and `end`.

```
  private int start, end;
```

11. Implement the constructor of the class that initializes its attributes.

```
  public Task(int array[],int start, int end) {
    this.array=array;
    this.start=start;
    this.end=end;
  }
```

12. Implement the `compute()` method to sum all the elements of the array between the start and end positions. First, convert the thread that is executing the task into a `MyWorkerThread` object and use the `addTask()` method to increment the counter of tasks for that thread.

```
  @Override
  protected Integer compute() {
    Integer ret;
    MyWorkerThread thread=(MyWorkerThread)Thread.currentThread();
    thread.addTask();
  }
```

13. Implement the `addResults()` method. Calculate and return the sum of the results of the two tasks received as parameters.

```
  private Integer addResults(Task task1, Task task2) {
    int value;
    try {
      value = task1.get().intValue()+task2.get().intValue();
    } catch (InterruptedException e) {
      e.printStackTrace();
      value=0;
    } catch (ExecutionException e) {
      e.printStackTrace();
      value=0;
    }
```

14. Put the thread to sleep for 10 milliseconds and return the result of the task.

```
    try {
      TimeUnit.MILLISECONDS.sleep(10);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return value;
  }
```

15. Implement the  main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

16. Create a `MyWorkerThreadFactory` object named `factory`.

```
    MyWorkerThreadFactory factory=new MyWorkerThreadFactory();
```

17. Create a `ForkJoinPool` object named `pool`. Pass to the constructor the factory object created earlier.

```
    ForkJoinPool pool=new ForkJoinPool(4, factory, null, false);
```

18. Create an array of 100,000 integers. Initialize all the elements to `1`.

```
    int array[]=new int[100000];

    for (int i=0; i<array.length; i++){
      array[i]=1;
    }
```

19. Create a new `Task` object to sum all the elements of the array.

```
    MyRecursiveTask task=new MyRecursiveTask(array,0,array.
length);
```

20. Send the task to the pool using the `execute()` method.

```
    pool.execute(task);
```

21. Wait for the end of the task using the `join()` method.

```
    task.join();
```

22. Shut down the pool using the `shutdown()` method.

    ```
    pool.shutdown();
    ```

23. Wait for the finalization of the executor using the `awaitTermination()` method.

    ```
    pool.awaitTermination(1, TimeUnit.DAYS);
    ```

24. Write in the console the result of the task using the `get()` method.

    ```
    System.out.printf("Main: Result: %d\n",task.get());
    ```

25. Write a message in the console indicating the end of the example.

    ```
    System.out.printf("Main: End of the program\n");
    ```

## How it works...

Threads used by the Fork/Join framework are called worker threads. Java includes the `ForkJoinWorkerThread` class that extends the `Thread` class and implements the worker threads used by the Fork/Join framework.

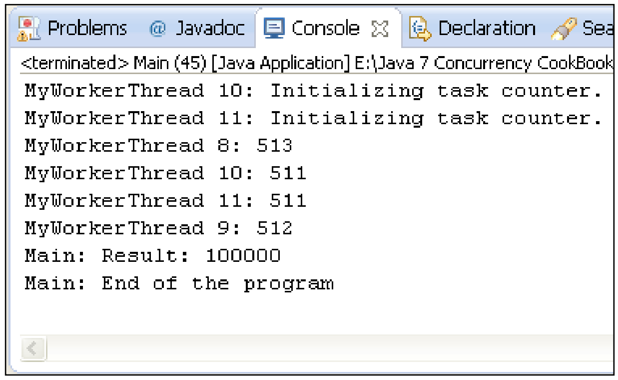
In this recipe, you have implemented the `MyWorkerThread` class that extends the `ForkJoinWorkerThread` class and overrides two methods of that class. Your objective is to implement a counter of tasks in each worker thread so you can know how many tasks a worker thread has executed. You have implemented the counter with a `ThreadLocal` attribute. This way, each thread will have its own counter in a transparent way for you, the programmer.

You have overridden the `onStart()` method of the `ForkJoinWorkerThread` class to initialize the task counter. This method is called when the worker thread begins its execution. You also have overridden the `onTermination()` method to print the value of the task counter to the console. This method is called when the worker thread finishes its execution. You have also implemented a method in the `MyWorkerThread` class. The `addTask()` method increments the task counter of each thread.

The `ForkJoinPool` class, as all the executors in the Java concurrency API, creates its threads using a factory, so if you want to use the `MyWorkerThread` threads in a `ForkJoinPool` class, you have to implement your thread factory. For the Fork/Join framework, this factory has to implement the `ForkJoinPool.ForkJoinWorkerThreadFactory` class. You have implemented the `MyWorkerThreadFactory` class for this purpose. This class only has one method that creates a new `MyWorkerThread` object.

Finally, you only have to initialize a `ForkJoinPool` class with the factory you have created. You have done this in the `Main` class, using the constructor of the `ForkJoinPool` class.

The following screenshot shows part of the output of the program:




You can see how the `ForkJoinPool` object has executed four worker threads and how many tasks have executed each of them.

## There's more...

Take into account that the `onTermination()` method provided by the `ForkJoinWorkerThread` class is called when a thread finishes normally or throws an `Exception` exception. The method receives a `Throwable` object as a parameter. If the parameter takes the `null` value, the worker thread finishes normally, but if the parameter takes a value, the thread throws an exception. You have to include the necessary code to process that situation.

## See also

- The *Create a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*
- The *Creating Threads through a factory* recipe in *Chapter 1*, *Thread Management*

# Customizing tasks running in the Fork/Join framework

The Executor framework separates the task creation and its execution. With it, you only have to implement the `Runnable` objects and use an `Executor` object. You send the `Runnable` tasks to the executor and it creates, manages, and finalizes the necessary threads to execute those tasks.

Java 7 provides a special kind of executor in the Fork/Join framework. This framework is designed to solve those problems that can be broken into smaller tasks using the divide and conquer technique. Inside a task, you have to check the size of the problem you want to resolve and, if it's bigger than an established size, you divide the problem in two or more tasks and execute those tasks using the framework. If the size of the problem is smaller than the established size, you resolve the problem directly in the task and then, optionally, it returns a result. The Fork/Join framework implements the work-stealing algorithm that improves the overall performance of these kinds of problems.

The main class of the Fork/Join framework is the `ForkJoinPool` class. Internally, it has the following two elements:

- A queue of tasks that are waiting to be executed
- A pool of threads that execute the tasks

By default, the tasks executed by a `ForkJoinPool` class are objects of the `ForkJoinTask` class. You also can send to a `ForkJoinPool` class the `Runnable` and `Callable` objects, but they can't take advantage of all the benefits of the Fork/Join framework. Normally, you will send to the `ForkJoinPool` objects one of two subclasses of the `ForkJoinTask` class:

- `RecursiveAction`: If your tasks don't return a result
- `RecursiveTask`: If your tasks return a result

In this recipe, you will learn how to implement your own tasks for the Fork/Join framework implementing a task that extends the `ForkJoinTask` class. The task you're going to implement measures and writes in the console its execution time, so you can control its evolution. You can also implement your own Fork/Join task to write log information, to get resources used in the tasks, or to post-process the results of the tasks.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyWorkerTask` and specify that it extends the `ForkJoinTask` class parameterized with the `Void` type.

   ```
   public abstract class MyWorkerTask extends ForkJoinTask<Void> {
   ```

2. Declare a private `String` attribute named `name` to store the name of the task.

   ```
     private String name;
   ```

3. Implement the constructor of the class to initialize its attribute.

   ```
     public MyWorkerTask(String name) {
       this.name=name;
     }
   ```

4. Implement the `getRawResult()` method. This is one of the abstract methods of the `ForkJoinTask` class. As the `MyWorkerTask` tasks won't return any result, this method must return the `null` value.

   ```
   @Override
   public Void getRawResult() {
     return null;
   }
   ```

5. Implement the `setRawResult()` method. This is another abstract method of the `ForkJoinTask` class. As the `MyWorkerTask` tasks won't return any result, leave the body of this method empty.

   ```
   @Override
   protected void setRawResult(Void value) {

   }
   ```

6. Implement the `exec()` method. This is the main method of the task. In this case, delegate the logic of the task to the `compute()` method. Calculate the execution time of that method and write it in the console.

   ```
   @Override
   protected boolean exec() {
     Date startDate=new Date();
     compute();
     Date finishDate=new Date();
     long diff=finishDate.getTime()-startDate.getTime();
     System.out.printf("MyWorkerTask: %s : %d Milliseconds to
   complete.\n",name,diff);
     return true;
   }
   ```

7. Implement the `getName()` method to return the name of the task.

   ```
   public String getName(){
     return name;
   }
   ```

8. Declare the abstract method `compute()`. As we mentioned earlier, this method will implement the logic of the tasks and must be implemented by the child classes of the `MyWorkerTask` class.

   ```
   protected abstract void compute();
   ```

9. Create a class named `Task` that extends the `MyWorkerTask` class.

   ```
   public class Task extends MyWorkerTask {
   ```

10. Declare a private array of `int` values named `array`.

    ```
    private int array[];
    ```

11. Implement a constructor of the class that initializes its attributes.

    ```
    public Task(String name, int array[], int start, int end){
      super(name);
      this.array=array;
      this.start=start;
      this.end=end;
    }
    ```

12. Implement the `compute()` method. This method increments the block of elements of the array determined by the start and end attributes. If this block of elements has more than 100 elements, divide the block in two parts and create two `Task` objects to process each part. Send those tasks to the pool using the `invokeAll()` method.

    ```
    protected void compute() {
      if (end-start>100){
        int mid=(end+start)/2;
        Task task1=new Task(this.getName()+"1",array,start,mid);
        Task task2=new Task(this.getName()+"2",array,mid,end);
        invokeAll(task1,task2);
    ```

13. If the block of elements has less than 100 elements, increment all the elements using a `for` loop.

    ```
      } else {
        for (int i=start; i<end; i++) {
          array[i]++;
        }
    ```

14. Finally, put the thread that is executing the task to sleep for 50 milliseconds.

    ```
        try {
          Thread.sleep(50);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
    ```

15. Implement the main class of the example by creating a class named `Main` with a `main()` method.

    ```
    public class Main {
      public static void main(String[] args) throws Exception {
    ```

16. Create an `int` array of 10,000 elements.

```
int array[]=new int[10000];
```

17. Create a `ForkJoinPool` object named `pool`.

```
ForkJoinPool pool=new ForkJoinPool();
```

18. Create a `Task` object to increment all the elements of the array. The parameters of the constructor are `Task` as the name of the task, the array object, and the values `0` and `10000` to indicate to this task that it has to process the entire array.

```
Task task=new Task("Task",array,0,array.length);
```

19. Send the task to the pool using the `execute()` method.

```
pool.invoke(task);
```

20. Shut down the pool using the `shutdown()` method.

```
pool.shutdown();
```

21. Write a message in the console indicating the end of the program.

```
System.out.printf("Main: End of the program.\n");
```

## How it works...

In this recipe, you have implemented the `MyWorkerTask` class that extends the `ForkJoinTask` class. It's your own base class to implement tasks that can be executed in a `ForkJoinPool` executor and that can take advantage of all the benefits of that executor, as the work-stealing algorithm. This class is equivalent to the `RecursiveAction` and `RecursiveTask` classes.

When you extend the `ForkJoinTask` class, you have to implement the following three methods:

- `setRawResult()`: This method is used to establish the result of the task. As your tasks don't return any results, you leave this method empty.
- `getRawResult()`: This method is used to return the result of the task. As your tasks don't return any results, this method returns the `null` value.
- `exec()`: This method implements the logic of the task. In your case, you have delegated the logic in the abstract method `compute()` (as the `RecursiveAction` and `RecursiveTask` classes) and in the `exec()` method you measure the execution time of that method, writing it in the console.

Finally, in the main class of the example, you have created an array of 10,000 elements, a `ForkJoinPool` executor, and a `Task` object to process the whole array. Execute the program and you'll see how the different tasks that are executed write their execution time in the console.

## See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*
- The *Implementing the ThreadFactory interface to generate custom threads for the Fork/Join framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Implementing a custom Lock class

Locks are one of the basic synchronization mechanisms provided by the Java concurrency API. It allows the programmers to protect a critical section of code, so only one thread can execute that block of code at a time. It provides the following two operations:

- `lock()`: You call this operation when you want to access a critical section. If there is another thread running that critical section, other threads are blocked until they're woken up by the lock to get the access to the critical section.
- `unlock()`: You call this operation at the end of the critical section, to allow other threads to access the critical section.

In the Java Concurrency API, locks are declared in the `Lock` interface and implemented in some classes, for example, the `ReentrantLock` class.

In this recipe, you will learn how to implement your own `Lock` object implementing a class that implements the `Lock` interface that can be used to protect a critical section.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyQueuedSynchronizer` that extends the `AbstractQueuedSynchronizer` class.

   ```
   public class MyAbstractQueuedSynchronizer extends
   AbstractQueuedSynchronizer {
   ```

2. Declare a private `AtomicInteger` attribute named `state`.

   ```
     private AtomicInteger state;
   ```

3. Implement the constructor of the class to initialize its attribute.

```
public MyAbstractQueuedSynchronizer() {
  state=new AtomicInteger(0);
}
```

4. Implement the `tryAcquire()` method. This method tries to change the value of the state variable from zero to one. If it can, it returns the `true` value else it returns `false`.

```
@Override
protected boolean tryAcquire(int arg) {
  return state.compareAndSet(0, 1);
}
```

5. Implement the `tryRelease()` method. This method tries to change the value of the state variable from one to zero. If it can, it returns the `true` value else it returns the `false` value.

```
@Override
protected boolean tryRelease(int arg) {
  return state.compareAndSet(1, 0);
}
```

6. Create a class named `MyLock` and specify that it implements the `Lock` interface.

```
public class MyLock implements Lock{
```

7. Declare a private `AbstractQueuedSynchronizer` attribute named `sync`.

```
private AbstractQueuedSynchronizer sync;
```

8. Implement the constructor of the class to initialize the `sync` attribute with a new `MyAbstractQueueSynchronizer` object.

```
public MyLock() {
  sync=new MyAbstractQueuedSynchronizer();
}
```

9. Implement the `lock()` method. Call the `acquire()` method of the `sync` object.

```
@Override
public void lock() {
  sync.acquire(1);
}
```

10. Implement the `lockInterruptibly()` method. Call the `acquireInterruptibly()` method of the `sync` object.

```
@Override
```

```
public void lockInterruptibly() throws InterruptedException {
  sync.acquireInterruptibly(1);
}
```

11. Implement the `tryLock()` method. Call the `tryAcquireNanos()` method of the `sync` object.

```
@Override
public boolean tryLock() {
  try {
    return sync.tryAcquireNanos(1, 1000);
  } catch (InterruptedException e) {
    e.printStackTrace();
    return false;
  }
}
```

12. Implement another version of the `tryLock()` method with two parameters. A long parameter named `time` and a `TimeUnit` parameter named `unit`. Call the `tryAcquireNanos()` method of the `sync` object.

```
@Override
public boolean tryLock(long time, TimeUnit unit)
    throws InterruptedException {
  return sync.tryAcquireNanos(1, TimeUnit.NANOSECONDS.
convert(time, unit));
}
```

13. Implement the `unlock()` method. Call the `release()` method of the `sync` object.

```
@Override
public void unlock() {
  sync.release(1);
}
```

14. Implement the `newCondition()` method. Create a new object of the internal class of the `sync` object `ConditionObject`.

```
@Override
public Condition newCondition() {
  return sync.new ConditionObject();
}
```

15. Create a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable {
```

16. Declare a private `MyLock` attribute named `lock`.

    ```
    private MyLock lock;
    ```

17. Declare a private `String` attribute named `name`.

    ```
    private String name;
    ```

18. Implement the constructor of the class to initialize its attributes.

    ```
    public Task(String name, MyLock lock){
      this.lock=lock;
      this.name=name;
    }
    ```

19. Implement the `run()` method of the class. Acquire the lock, put the thread to sleep for 2 seconds and then, release the `lock` object.

    ```
    @Override
    public void run() {
      lock.lock();
      System.out.printf("Task: %s: Take the lock\n",name);
      try {
        TimeUnit.SECONDS.sleep(2);
        System.out.printf("Task: %s: Free the lock\n",name);
      } catch (InterruptedException e) {
        e.printStackTrace();
      } finally {
        lock.unlock();
      }
    }
    ```

20. Implement the main class of the example by creating a class named `Main` with a `main()` method.

    ```
    public class Main {
      public static void main(String[] args) {
    ```

21. Create a `MyLock` object named `lock`.

    ```
        MyLock lock=new MyLock();
    ```

22. Create and execute 10 `Task` tasks.

    ```
        for (int i=0; i<10; i++){
          Task task=new Task("Task-"+i,lock);
          Thread thread=new Thread(task);
          thread.start();
        }
    ```

23. Try to get the lock using the `tryLock()` method. Wait for a second and if you don't get the lock, write a message and try again.

```
boolean value;
do {
  try {
    value=lock.tryLock(1,TimeUnit.SECONDS);
    if (!value) {
      System.out.printf("Main: Trying to get the Lock\n");
    }
  } catch (InterruptedException e) {
    e.printStackTrace();
    value=false;
  }
} while (!value);
```

24. Write a message indicating that you got the lock and release it.

```
System.out.printf("Main: Got the lock\n");
lock.unlock();
```

25. Write a message indicating the end of the program.

```
System.out.printf("Main: End of the program\n");
```

## How it works...

Java Concurrency API provides a class that can be used to implement synchronization mechanisms with features of locks or semaphores. It's `AbstractQueuedSynchronizer` and, as its name suggests, it's an abstract class. It provides operations to control access to a critical section and to manage a queue of threads that are blocked awaiting access to the critical section. The operations are based on two abstract methods:

- `tryAcquire()`: This method is called to try to get access to a critical section. If the thread that calls it can access the critical section, the method returns the `true` value. Otherwise, the method returns the `false` value.
- `tryRelease()`: This method is called to try to release access to a critical section. If the thread that calls it can release the access, the method returns the `true` value. Else, the method returns the `false` value.

In these methods, you have to implement the mechanism you use to control the access to the critical section. In your case, you have implemented the `MyQueuedSynchonizer` class that extends the `AbstractQueuedSyncrhonizer` class and implements the abstract methods using an `AtomicInteger` variable to control the access of the critical section. That variable will store the `0` value if the lock is free, so a thread can have access to the critical section, and the `1` value if the lock is blocked, so a thread can't have access to the critical section.

You have used the `compareAndSet()` method provided by the `AtomicInteger` class that tries to change the value you specify as the first parameter for the value you specify as the second parameter. To implement the `tryAcquire()` method, you try to change the value of the atomic variable from zero to one. Similarly, to implement the `tryRelease()` method, you try to change the value of the atomic variable from one to zero.

You have to implement this class because other implementations of the `AbstractQueuedSynchronizer` class (for example, the one used by the `ReentrantLock` class), are implemented as private classes internally in the class that uses it, so you don't have access to it.

Then, you have implemented the `MyLock` class. This class implements the `Lock` interface and has a `MyQueuedSynchronizer` object as an attribute. To implement all the methods of the `Lock` interface, you have used methods of the `MyQueuedSynchronizer` object.

Finally, you have implemented the `Task` class, that implements the `Runnable` interface and uses a `MyLock` object to get the access to a critical section. That critical section puts the thread to sleep for 2 seconds. The main class creates a `MyLock` object and runs 10 `Task` objects that share that lock. The main class also tries to get the access to the lock using the `tryLock()` method.

When you execute the example, you can see how only one thread has access to the critical section and when that thread finishes, another one gets the access to it.

You can use your own `Lock` to write log messages about its utilization, control the time that is locked, or implement advanced synchronization mechanisms, to control, for example, the access to a resource so that it's only available at certain times.

## There's more...

The `AbstractQueuedSynchronizer` class provides two methods that can be used to manage the state of the lock. They are the `getState()` and `setState()` methods. These methods receive and return an integer value with the state of the lock. You could have used those methods instead of the `AtomicInteger` attribute to store the state of the lock.

Java concurrency API provides another class to implement synchronization mechanisms. It's the `AbstractQueuedLongSynchronizer` class, that is equivalent to the `AbstractQueuedSynchronizer` class, but uses a `long` attribute to store the state of the threads.

## See also

- The *Synchronizing a block of code with locks* recipe in *Chapter 2, Basic Thread Synchronization*

# Implementing a transfer Queue based on priorities

Java 7 API provides several data structures to work with concurrent applications. From these, we want to highlight the following two data structures:

- `LinkedTransferQueue`: This data structure is supposed to be used in those programs that have a producer/consumer structure. In those applications, you have one or more producers of data and one or more consumers of data and a data structure is shared by all of them. The producers put data in the data structure and the consumers take data from the data structure. If the data structure is empty, the consumers are blocked until they have data to consume. If the data structure is full, the producers are blocked until they have space to put their data.
- `PriorityBlockingQueue`: In this data structure, elements are stored in an ordered way. The elements have to implement the `Comparable` interface with the `compareTo()` method. When you insert an element in the structure, it's compared to the elements of the structure until it finds its position.

Elements of the `LinkedTransferQueue` are stored in the same order as they arrive, so earlier arrivals are consumed first. It may be the case when you want to develop a producer/consumer program, where data is consumed according to some priority instead of arrival time. In this recipe, you will learn how to implement a data structure to be used in the producer/consumer problem, whose elements will be ordered by their priority. Those elements with higher priority will be consumed first.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyPriorityTransferQueue` that extends the `PriorityBlockingQueue` class and implements the `TransferQueue` interface.

```
public class MyPriorityTransferQueue<E> extends
PriorityBlockingQueue<E> implements
    TransferQueue<E> {
```

2. Declare a private `AtomicInteger` attribute named `counter` to store the number of consumers that are waiting for elements to consume.

   ```
   private AtomicInteger counter;
   ```

3. Declare a private `LinkedBlockingQueue` attribute named `transferred`.

   ```
   private LinkedBlockingQueue<E> transfered;
   ```

4. Declare a private `ReentrantLock` attribute named `lock`.

   ```
   private ReentrantLock lock;
   ```

5. Implement the constructor of the class to initialize its attributes.

   ```
   public MyPriorityTransferQueue() {
     counter=new AtomicInteger(0);
     lock=new ReentrantLock();
     transfered=new LinkedBlockingQueue<E>();
   }
   ```

6. Implement the `tryTransfer()` method. This method tries to send the element to a waiting consumer immediately, if possible. If there isn't any waiting consumer, the method returns the `false` value.

   ```
   @Override
   public boolean tryTransfer(E e) {
     lock.lock();
     boolean value;
     if (counter.get()==0) {
       value=false;
     } else {
       put(e);
       value=true;
     }
     lock.unlock();
     return value;
   }
   ```

7. Implement the `transfer()` method. This method tries to send the element to a waiting consumer immediately, if possible. If there isn't a waiting consumer, the method stores the element in a special queue to be sent to the first consumer that tries to get an element and blocks the thread until the element is consumed.

   ```
   @Override
   public void transfer(E e) throws InterruptedException {
     lock.lock();
     if (counter.get()!=0) {
       put(e);
       lock.unlock();
   ```

```
    } else {
      transfered.add(e);
      lock.unlock();
      synchronized (e) {
        e.wait();
      }
    }
  }
```

8. Implement the `tryTransfer()` method that receives three parameters: The element, the time to wait for a consumer, if there is none, and the unit of time used to specify that time. If there is a consumer waiting, it sends the element immediately. Otherwise, convert the time specified to milliseconds and use the `wait()` method to put the thread to sleep. When the consumer takes the element, if the thread is sleeping in the `wait()` method, you are going to wake it up using the `notify()` method as you'll see in a moment.

```
  @Override
  public boolean tryTransfer(E e, long timeout, TimeUnit unit)
      throws InterruptedException {
    lock.lock();
    if (counter.get()!=0) {
      put(e);
      lock.unlock();
      return true;
    } else {
      transfered.add(e);
      long newTimeout= TimeUnit.MILLISECONDS.convert(timeout, 
unit);
      lock.unlock();
      e.wait(newTimeout);
      lock.lock();
      if (transfered.contains(e)) {
        transfered.remove(e);
        lock.unlock();
        return false;
      } else {
        lock.unlock();
        return true;
      }
    }
  }
```

9. Implement the `hasWaitingConsumer()` method. Use the value of the counter attribute to calculate the return value of this method. If the counter has a value bigger than zero, return `true`. Else, return `false`.

```
@Override
public boolean hasWaitingConsumer() {
  return (counter.get()!=0);
}
```

10. Implement the `getWaitingConsumerCount()` method. Return the value of the `counter` attribute.

```
@Override
public int getWaitingConsumerCount() {
  return counter.get();
}
```

11. Implement the `take()` method. This `method` is called by the consumers when they want an element to consume. First, get the lock defined earlier and increment the number of waiting consumers.

```
@Override
public E take() throws InterruptedException {
  lock.lock();
  counter.incrementAndGet();
```

12. If there aren't any elements in the transferred queue, free the lock and try to get an element from the queue using the `take()` element and get the lock again. If there aren't any elements in the queue, this method will put the thread to sleep until there are elements to consume.

```
  E value=transfered.poll();
  if (value==null) {
    lock.unlock();
    value=super.take();
    lock.lock();
```

13. Otherwise, take the element from the transferred queue and wake up the thread that is waiting for the consummation of that element, if there is one.

```
  } else {
    synchronized (value) {
      value.notify();
    }
  }
```

14. Finally, decrement the counter of waiting consumers and free the lock.

```
  counter.decrementAndGet();
  lock.unlock();
```

```
    return value;
  }
```

15. Implement a class named `Event` that extends the `Comparable` interface parameterized with the `Event` class.

```
public class Event implements Comparable<Event> {
```

16. Declare a private `String` attribute named `thread` to store the name of the thread that creates the event.

```
  private String thread;
```

17. Declare a private `int` attribute named `priority` to store the priority of the event.

```
  private int priority;
```

18. Implement the constructor of the class to initialize its attributes.

```
  public Event(String thread, int priority){
    this.thread=thread;
    this.priority=priority;
  }
```

19. Implement a method to return the value of the thread attribute.

```
  public String getThread() {
    return thread;
  }
```

20. Implement a method to return the value of the priority attribute.

```
  public int getPriority() {
    return priority;
  }
```

21. Implement the `compareTo()` method. This method compares the actual event with an event received as a parameter. Return `-1` if the actual event has a bigger priority than the parameter, `1` if the actual event has a lower priority than the parameter, and `0` if both events have the same priority. You will get the list ordered by priority in the descending order. Events with higher priority will be stored first in the queue.

```
  public int compareTo(Event e) {
    if (this.priority>e.getPriority()) {
      return -1;
    } else if (this.priority<e.getPriority()) {
      return 1;
    } else {
      return 0;
    }
  }
```

22. Implement a class named `Producer` that implements the `Runnable` interface.

    ```
    public class Producer implements Runnable {
    ```

23. Declare a private `MyPriorityTransferQueue` attribute parameterized with the `Event` class named `buffer` to store the events generated by this producer.

    ```
      private MyPriorityTransferQueue<Event> buffer;
    ```

24. Implement the constructor of the class to initialize its attributes.

    ```
      public Producer(MyPriorityTransferQueue<Event> buffer) {
        this.buffer=buffer;
      }
    ```

25. Implement the `run()` method of the class. Create 100 `Event` objects using its order of creation as priority (the latest event will have the highest priority) and insert them in the queue using the `put()` method.

    ```
      @Override
      public void run() {
        for (int i=0; i<100; i++) {
          Event event=new Event(Thread.currentThread().getName(),i);
          buffer.put(event);
        }
      }
    ```

26. Implement a class named `Consumer` that implements the `Runnable` interface.

    ```
    public class Consumer implements Runnable {
    ```

27. Declare a private `MyPriorityTransferQueue` attribute parameterized with the `Event` class named buffer to get the events consumed by this class.

    ```
      private MyPriorityTransferQueue<Event> buffer;
    ```

28. Implement the constructor of the class to initialize its attribute.

    ```
      public Consumer(MyPriorityTransferQueue<Event> buffer) {
        this.buffer=buffer;
      }
    ```

29. Implement the `run()` method. It consumes 1002 `Events` (all the events generated in the example) using the `take()` method and write the number of the thread that generated the event and its priority in the console.

    ```
      @Override
      public void run() {
        for (int i=0; i<1002; i++) {
          try {
            Event value=buffer.take();
    ```

```
        System.out.printf("Consumer: %s: %d\n",value.
getThread(),value.getPriority());
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

30. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

31. Create a `MyPriorityTransferQueue` object named `buffer`.

```
    MyPriorityTransferQueue<Event> buffer=new MyPriorityTransferQu
eue<Event>();
```

32. Create a `Producer` task and launch 10 threads to execute that task.

```
    Producer producer=new Producer(buffer);

    Thread producerThreads[]=new Thread[10];
    for (int i=0; i<producerThreads.length; i++) {
      producerThreads[i]=new Thread(producer);
      producerThreads[i].start();
    }
```

33. Create and launch a `Consumer` task.

```
    Consumer consumer=new Consumer(buffer);
    Thread consumerThread=new Thread(consumer);
    consumerThread.start();
```

34. Write in the console the actual consumer count.

```
    System.out.printf("Main: Buffer: Consumer count: %d\n",buffer.
getWaitingConsumerCount());
```

35. Transfer an event to the consumer using the `transfer()` method.

```
    Event myEvent=new Event("Core Event",0);
    buffer.transfer(myEvent);
    System.out.printf("Main: My Event has ben transfered.\n");
```

36. Wait for the finalization of the producers using the `join()` method.

```
    for (int i=0; i<producerThreads.length; i++) {
      try {
        producerThreads[i].join();
```

```
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
```

37. Put the thread to sleep for 1 second.

```
        TimeUnit.SECONDS.sleep(1);
```

38. Write the actual consumer count.

```
      System.out.printf("Main: Buffer: Consumer count: %d\n",buffer.
getWaitingConsumerCount());
```

39. Transfer another event using the `transfer()` method.

```
      myEvent=new Event("Core Event 2",0);
        buffer.transfer(myEvent);
```

40. Wait for the finalization of the consumer using the `join()` method.

```
        consumerThread.join();
```

41. Write a message indicating the end of the program.

```
      System.out.printf("Main: End of the program\n");
```

## How it works...

In this recipe, you have implemented the `MyPriorityTransferQueue` data structure. It's a data structure to be used in the producer/consumer problem, but its elements are ordered by priority, not by their arrival order. As Java doesn't allow multiple inheritance, the first decision you have taken is the base class of the `MyPriorityTransferQueue` class. You have extended the `PriorityBlockingQueue` class, to have implemented the operations that insert the elements in the structure ordered by priority. You also have implemented the `TransferQueue` interface to add the methods related with the producer/consumer.

The `MyPriortyTransferQueue` class have the following three attributes:

- An `AtomicInteger` attribute, named `counter`: This attribute stores the number of consumers that are waiting for taking an element for the data structure. When a consumer calls the `take()` operation to take an element from the data structure, the counter is incremented. When the consumer finishes the execution of the `take()` operation, the counter is decremented again. This counter is used in the implementation of the `hasWaitingConsumer()` and `getWaitingConsumerCount()` methods.

- A `ReentrantLock` attribute named `lock`: This attribute is used to control the access to the implemented operations. Only one thread can be working with the data structure.
- Finally, a `LinkedBlockingQueue` list to store the transferred elements.

You have implemented some methods in the `MyPriorityTransferQueue`. All the methods are declared in the `TransferQueue` interface and the `take()` method implemented in the `PriorityBlockingQueue` interface. Two of them were described before. Here is a description of the rest:

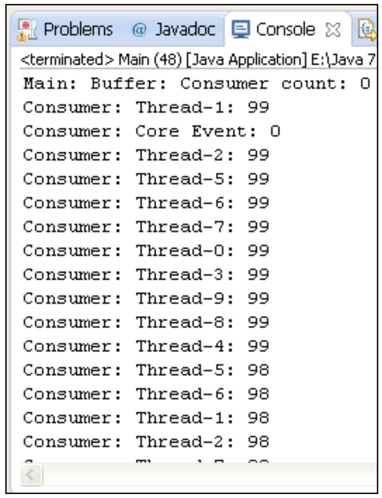
- `tryTransfer(E e)`: This method tries to send an element directly to a consumer. If there is a consumer waiting, the method stores the element in the priority queue to be consumed immediately by the consumer and then returns the `true` value. If there isn't a consumer waiting, the method returns the `false` value.
- `transfer(E e)`: This method transfers an element directly to a consumer. It there is a consumer waiting, the method stores the element in the priority queue to be consumed immediately by the consumer. Otherwise, the element is stored in the list for transferred elements and the thread is blocked until the element is consumed. While the thread is put to sleep, you have to free the lock because if not, you block the queue.
- `tryTransfer(E e, long timeout, TimeUnit unit)`: This method is similar to the `transfer()` method, but the thread blocks the period of time determined by its parameters. While the thread is put to sleep, you have to free the lock because, if not, you block the queue.
- `take()`: This method returns the next element to be consumed. If there are elements in the list of transferred elements, the element to be consumed is taken from that list. Otherwise, it is taken from the priority queue.

Once you have implemented the data structure, you have implemented the `Event` class. It is the class of the elements you have stored in the data structure. The `Event` class has two attributes to store the ID of the producer and the priority of the event, and implements the `Comparable` interface, because it is a requirement of your data structure.

Then, you have implemented the `Producer` and the `Consumer` classes. In the example, you have 10 producers and a consumer and they share the same buffer. Each producer generates 100 events with incremental priority, so the events with higher priority are the last generated ones.

The main class of example creates a `MyPriorityTransferQueue` object, 10 producers, and a consumer and uses the `transfer()` method of the `MyPriorityTransferQueue` buffer to transfer two events to the buffer.

The following screenshot shows part of the output of an execution of the program:




You can see how the events with higher priority are consumed first, and that a consumer consumes the transferred event.

## See also

- The *Using blocking thread-safe lists ordered by priority* recipe in *Chapter 6*, *Concurrent Collections*
- The *Using blocking thread-safe lists* recipe in *Chapter 6*, *Concurrent Collections*

# Implementing your own atomic object

Atomic variables were introduced in Java Version 5 and provide atomic operations on single variables. When a thread is doing an operation with an atomic variable, the implementation of the class includes a mechanism to check that the operation is done in one step. Basically, the operation gets the value of the variable, changes the value in a local variable, and then tries to change the old value for the new one. If the old value is still the same, it does the change. If not, the method begins the operation again.

In this recipe, you will learn how to extend an atomic object and how to implement two operations that follow the mechanisms of the atomic objects to guarantee that all the operations are done in one step.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `ParkingCounter` and specify that it extends the `AtomicInteger` class.

   ```
   public class ParkingCounter extends AtomicInteger {
   ```

2. Declare a private `int` attribute named `maxNumber` to store the maximum number of cars admitted in the parking lot.

   ```
     private int maxNumber;
   ```

3. Implement the constructor of the class to initialize its attributes.

   ```
     public ParkingCounter(int maxNumber){
       set(0);
       this.maxNumber=maxNumber;
     }
   ```

4. Implement the `carIn()` method. This method increments the counter of cars if it has a value smaller than the established maximum value. Construct an infinite loop and get the value of the internal counter using the `get()` method.

   ```
     public boolean carIn() {
       for (;;) {
         int value=get();
   ```

5. If the value is equal to the `maxNumber` attribute, the counter can't be incremented (the parking lot is full and the car can't enter). The method returns the `false` value.

   ```
         if (value==maxNumber) {
           System.out.printf("ParkingCounter: The parking lot is 
   full.\n");
           return false;
   ```

6. Otherwise, increment the value and use the `compareAndSet()` method to change the old value to the new one. This method returns the `false` value; the counter was not incremented, so you have to begin the loop again. If it returns the `true` value, it means the change was made and then, you return the `true` value.

```
      } else {
        int newValue=value+1;
        boolean changed=compareAndSet(value,newValue);
        if (changed) {
          System.out.printf("ParkingCounter: A car has
entered.\n");
          return true;
        }
      }
    }
  }
```

7. Implement the `carOut()` method. This method decrements the counter of cars if it has a value bigger than `0`. Construct an infinite loop and get the value of the internal counter using the `get()` method.

```
  public boolean carOut() {
    for (;;) {
      int value=get();
      if (value==0) {
        System.out.printf("ParkingCounter: The parking lot is
        empty.\n");
        return false;
      } else {
        int newValue=value-1;
        boolean changed=compareAndSet(value,newValue);
        if (changed) {
        System.out.printf("ParkingCounter: A car has gone
        out.\n");
          return true;
        }
      }
    }
  }
```

8. Create a class named `Sensor1` that implements the `Runnable` interface.

```
public class Sensor1 implements Runnable {
```

9. Declare a private `ParkingCounter` attribute named `counter`.

```
  private ParkingCounter counter;
```

10. Implement the constructor of the class to initialize its attribute.

```
public Sensor1(ParkingCounter counter) {
  this.counter=counter;
}
```

11. Implement the `run()` method. Call the `carIn()` and `carOut()` operations several times.

```
 @Override
public void run() {
  counter.carIn();
  counter.carIn();
  counter.carIn();
  counter.carIn();
  counter.carOut();
  counter.carOut();
  counter.carOut();
  counter.carIn();
  counter.carIn();
  counter.carIn();
}
```

12. Create a class named `Sensor2` that implements the `Runnable` interface.

```
public class Sensor2 implements Runnable {
```

13. Declare a private `ParkingCounter` attribute named `counter`.

```
private ParkingCounter counter;
```

14. Implement the constructor of the class to initialize its attribute.

```
public Sensor2(ParkingCounter counter) {
  this.counter=counter;
}
```

15. Implement the `run()` method. Call the `carIn()` and `carOut()` operations several times.

```
 @Override
public void run() {
  counter.carIn();
  counter.carOut();
  counter.carOut();
  counter.carIn();
  counter.carIn();
  counter.carIn();
  counter.carIn();
  counter.carIn();
  counter.carIn();
}
```

16. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

17. Create a `ParkingCounter` object named `counter`.

```
    ParkingCounter counter=new ParkingCounter(5);
```

18. Create and launch a `Sensor1` task and a `Sensor2` task.

```
    Sensor1 sensor1=new Sensor1(counter);
    Sensor2 sensor2=new Sensor2(counter);

    Thread thread1=new Thread(sensor1);
    Thread thread2=new Thread(sensor2);

    thread1.start();
    thread2.start();
```

19. Wait for the finalization of both tasks.

```
    thread1.join();
    thread2.join();
```

20. Write in the console the actual value of the counter.

```
    System.out.printf("Main: Number of cars: %d\n",counter.get());
```

21. Write in the console a message indicating the end of the program.

```
    System.out.printf("Main: End of the program.\n");
```

## How it works...

The `ParkingCounter` class extends the `AtomicInteger` class with two atomic operations, `carIn()` and `carOut()`. The example simulates a system that controls the number of cars inside a parking lot. The parking lot can admit a number of cars, represented by the `maxNumber` attribute.

The `carIn()` operation compares the actual number of cars in the parking lot with the maximum value. If they are equal, the car can't enter the parking lot and the method returns the `false` value. Otherwise, it uses the following structure of the atomic operations:

1. Get the value of the atomic object in a local variable.
2. Store the new value in a different variable.

3. Use the `compareAndSet()` method to try to replace the old value by the new one. If this method returns the `true` value, the old value you sent as a parameter was the value of the variable, so it makes the change of values. The operation was made in an atomic way as the `carIn()` method returns the `true` value. If the `compareAndSet()` method returns the `false` value, the old value you sent as a parameter is not the value of the variable (the other thread modified it), so the operation can't be done in an atomic way. The operation begins again until it can be done in an atomic way.

The `carOut()` method is analogous to the `carIn()` method. You have also implemented two `Runnable` objects that use the `carIn()` and `carOut()` methods to simulate the activity of the parking. When you execute the program, you can see that the parking lot never overcomes the maximum value of cars in the parking lot.

## See also

- The *Using atomic variables* recipe in *Chapter 6*, *Concurrent Collections*

# 8
# Testing Concurrent Applications

In this chapter, we will cover:

- Monitoring a `Lock` interface
- Monitoring a `Phaser` class
- Monitoring an Executor framework
- Monitoring a Fork/Join pool
- Writing effective log messages
- Analyzing concurrent code with FindBugs
- Configuring Eclipse for debugging concurrency code
- Configuring NetBeans for debugging concurrency code
- Testing concurrency code with MultithreadedTC

## Introduction

Testing an application is a critical task. Before the application is ready for end users, you have to demonstrate its correctness. You use a test process to prove that correctness is achieved and errors are fixed. The testing phase is a common task in any software development and also **quality assurance** processes. You can find a lot of literature about testing processes and the different approaches you can apply to your developments. There are also a lot of libraries, such as `JUnit`, and applications, such as Apache `JMetter` that you can use to test your Java applications in an automated way. It's even more critical in a concurrent application development.

The fact that concurrent applications have two or more threads that share data structures and interact with each other adds more difficulty to the testing phase. The biggest problem you will face when you test concurrent applications is that the execution of threads is non-deterministic. You can't guarantee the order of the execution of the threads, so it's difficult to reproduce errors.

In this chapter, you will learn:

- How to obtain information about the elements you have in your concurrent applications. This information can help you test your concurrent applications.
- How to use an IDE (Integrated Development Environment) and other tools such as FindBugs to test your concurrent applications.
- How to use libraries such as MultithreadedTC to automate your tests.

# Monitoring a Lock interface

A `Lock` interface is one of the basic mechanisms provided by the Java concurrency API to get the synchronization of a block of code. It allows the definition of a **critical section**. A critical section is a block of code that accesses a shared resource and can't be executed by more than one thread at the same time. This mechanism is implemented by the `Lock` interface and the `ReentrantLock` class.

In this recipe, you will learn what information you can obtain about a `Lock` object and how to obtain that information.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyLock` that extends the `ReentrantLock` class.

   ```
   public class MyLock extends ReentrantLock {
   ```

2. Implement the `getOwnerName()` method. This method returns the name of the thread that has the control of a lock (if any) using the protected method of the `Lock` class `getOwner()`.

   ```
     public String getOwnerName() {
       if (this.getOwner()==null) {
         return "None";
   ```

```
  }
  return this.getOwner().getName();
}
```

3. Implement the `getThreads()` method. This method returns a list of threads queued in a lock using the protected method of the `Lock` class `getQueuedThreads()`.

```
public Collection<Thread> getThreads() {
  return this.getQueuedThreads();
}
```

4. Create a class named `Task` that implements the `Runnable` interface.

```
public class Task implements Runnable {
```

5. Declare a private `Lock` attribute named `lock`.

```
  private Lock lock;
```

6. Implement a constructor of the class to initialize its attribute.

```
  public Task (Lock lock) {
    this.lock=lock;
  }
```

7. Implement the `run()` method. Create a loop with five steps.

```
  @Override
  public void run() {
    for (int i=0; i<5; i++) {
```

8. Acquire the lock using the `lock()` method and print a message.

```
      lock.lock();
      System.out.printf("%s: Get the Lock.\n",Thread.
currentThread().getName());
```

9. Put the thread to sleep for 500 milliseconds. Free the lock using the `unlock()` method and print a message.

```
      try {
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.printf("%s: Free the Lock.\n",Thread.
currentThread().getName());
      } catch (InterruptedException e) {
        e.printStackTrace();
      } finally {
        lock.unlock();
      }
        }
  }
}
```

10. Create the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

11. Create a `MyLock` object named `lock`.

```
    MyLock lock=new MyLock();
```

12. Create an array for five `Thread` objects.

```
    Thread threads[]=new Thread[5];
```

13. Create and start five threads to execute five `Task` objects.

```
    for (int i=0; i<5; i++) {
      Task task=new Task(lock);
      threads[i]=new Thread(task);
      threads[i].start();
    }
```

14. Create a loop with 15 steps.

```
    for (int i=0; i<15; i++) {
```

15. Write in the console the name of the owner of the lock.

```
      System.out.printf("Main: Logging the Lock\n");
      System.out.printf("************************\n");
      System.out.printf("Lock: Owner : %s\n",lock.getOwnerName());
```

16. Display the number and the name of the threads queued for the lock.

```
.out.printf("Lock: Queued Threads: %s\n",lock.hasQueuedThreads());
      if (lock.hasQueuedThreads()){
        System.out.printf("Lock: Queue Length: %d\n",lock.
getQueueLength());
        System.out.printf("Lock: Queued Threads: ");
        Collection<Thread> lockedThreads=lock.getThreads();
        for (Thread lockedThread : lockedThreads) {
        System.out.printf("%s ",lockedThread.getName());
        }
        System.out.printf("\n");
      }
```

17. Display information about the fairness and the status of the `Lock` object.

```
        System.out.printf("Lock: Fairness: %s\n",lock.isFair());
        System.out.printf("Lock: Locked: %s\n",lock.isLocked());
        System.out.printf("************************\n");
```

18. Put the thread to sleep for 1 second and close the loop and the class.

```
        TimeUnit.SECONDS.sleep(1);
      }

    }

  }
```

## How it works...

In this recipe, you have implemented the `MyLock` class that extends the `ReentrantLock` class to return information that otherwise wouldn't have been available – it's protected data of the `ReentrantLock` class. The methods implemented by the `MyLock` class are:

- `getOwnerName()`: Only one thread can execute a critical section protected by a `Lock` object. The lock stores the thread that is executing the critical section. This thread is returned by the protected `getOwner()` method of the `ReentrantLock` class. This method uses the `getOwner()` method to return the name of that thread.
- `getThreads()`: While a thread is executing a critical section, the other threads that try to enter it are put to sleep until they can continue executing that critical section. The protected method `getQueuedThreads()` of the `ReentrantLock` class returns the list of threads that are waiting to execute the critical section. This method returns the result returned by the `getQueuedThreads()` method.

We have also used other methods that are implemented in the `ReentrantLock` class:

- `hasQueuedThreads()`: This method returns a `Boolean` value indicating if there are threads waiting to acquire this lock
- `getQueueLength()`: This method returns the number of threads that are waiting to acquire this lock
- `isLocked()`: This method returns a `Boolean` value indicating whether this lock is owned by a thread
- `isFair()`: This method returns a `Boolean` value indicating if this lock has the fair mode activated

## There's more...

There are other methods in the `ReentrantLock` class that can be used to obtain information about a `Lock` object:

- `getHoldCount()`: Returns the number of times that the current thread has acquired the lock
- `isHeldByCurrentThread()`: Returns a `Boolean` value indicating if the lock is owned by the current thread

## See also

- The *Synchronizing a block of code with a lock* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Implementing a custom Lock class* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Monitoring a Phaser class

One of the most complex and powerful functionalities offered by the Java Concurrency API is the ability to execute concurrent phased tasks using the `Phaser` class. This mechanism is useful when we have some concurrent tasks divided in steps. The `Phaser` class provides us the mechanism to synchronize the threads at the end of each step, so no thread starts its second step until all the threads have finished the first one.

In this recipe, you will learn what information about the status of a `Phaser` class you can obtain and how to obtain that information.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` that implements the `Runnable` interface.

```
public class Task implements Runnable {
```

2. Declare a private `int` attribute named `time`.

```
  private int time;
```

3. Declare a private `Phaser` attribute named `phaser`.

```
private Phaser phaser;
```

4. Implement the constructor of the class to initialize its attributes.

```
public Task(int time, Phaser phaser) {
  this.time=time;
  this.phaser=phaser;
}
```

5. Implement the `run()` method. First, instruct the `phaser` attribute that the task starts its execution with the `arrive()` method.

```
  @Override
public void run() {

  phaser.arrive();
```

6. Write a message in the console indicating the start of phase one, put the thread to sleep for the number of seconds specified by the `time` attribute, write in the console a message indicating the end of phase one, and synchronize with the rest of the tasks using the `arriveAndAwaitAdvance()` method of the `phaser` attribute.

```
    System.out.printf("%s: Entering phase 1.\n",Thread.
currentThread().getName());
    try {
      TimeUnit.SECONDS.sleep(time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("%s: Finishing phase 1.\n",Thread.
currentThread().getName());
    phaser.arriveAndAwaitAdvance();
```

7. Repeat the behavior for the second and third phases. At the end of the third phase, use the `arriveAndDeregister()` method instead of `arriveAndAwaitAdvance()`.

```
    System.out.printf("%s: Entering phase 2.\n",Thread.
currentThread().getName());
    try {
      TimeUnit.SECONDS.sleep(time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("%s: Finishing phase 2.\n",Thread.
currentThread().getName());
    phaser.arriveAndAwaitAdvance();
```

```
    System.out.printf("%s: Entering phase 3.\n",Thread.
currentThread().getName());
    try {
      TimeUnit.SECONDS.sleep(time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("%s: Finishing phase 3.\n",Thread.
currentThread().getName());

    phaser.arriveAndDeregister();
```

8. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

9. Create a new `Phaser` object named `phaser` with three participants.

```
    Phaser phaser=new Phaser(3);
```

10. Create and launch three threads to execute three task objects.

```
    for (int i=0; i<3; i++) {
      Task task=new Task(i+1, phaser);
      Thread thread=new Thread(task);
      thread.start();
    }
```

11. Create a loop with 10 steps to write information about the `phaser` object.

```
    for (int i=0; i<10; i++) {
```

12. Write information about the registered parties, the phase of the phaser, the arrived parties, and the un-arrived parties.

```
    for (int i=0; i<10; i++) {
      System.out.printf("********************\n");
      System.out.printf("Main: Phaser Log\n");
      System.out.printf("Main: Phaser: Phase: %d\n",phaser.
getPhase());
      System.out.printf("Main: Phaser: Registered Parties:
%d\n",phaser.getRegisteredParties());
      System.out.printf("Main: Phaser: Arrived Parties:
%d\n",phaser.getArrivedParties());
      System.out.printf("Main: Phaser: Unarrived Parties:
%d\n",phaser.getUnarrivedParties());
      System.out.printf("********************\n");
```

13. Put the thread to sleep for 1 second and close the loop and the class.

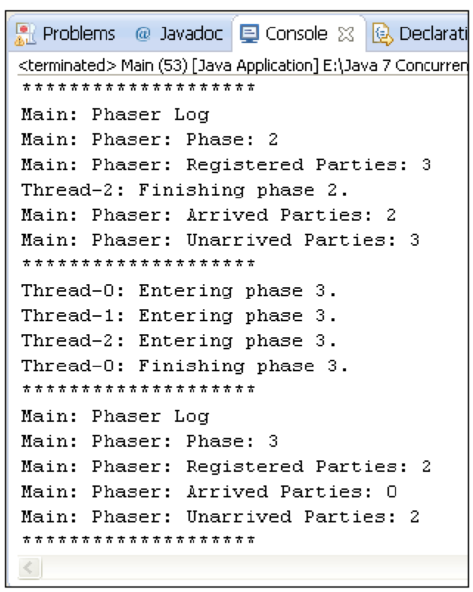
```
                TimeUnit.SECONDS.sleep(1);
            }
        }
    }
```

## How it works...

In this recipe, we have implemented a phased task in the `Task` class. This phased task has three phases and uses a `Phaser` interface to synchronize with other `Task` objects. The main class launches three tasks and while these tasks are executing their phases, it prints information about the status of the `phaser` object to the console. We have used the following methods to get the status of the `phaser` object:

- `getPhase()`: This method returns the actual phase of a `phaser` object
- `getRegisteredParties()`: This method returns the number of tasks that use a `phaser` object as a mechanism of synchronization
- `getArrivedParties()`: This method returns the number of tasks that have arrived at the end of the actual phase
- `getUnarrivedParties()`: This method returns the number of tasks that haven't yet arrived at the end of the actual phase

The following screenshot shows part of the output of the program:



## See also

- The *Running concurrent phased tasks* recipe in *Chapter 3*, *Thread Synchronization Utilities*

# Monitoring an Executor framework

The Executor framework provides a mechanism that separates the implementation of tasks from the thread creation and management to execute those tasks. If you use an executor, you only have to implement the `Runnable` objects and send them to the executor. It is the responsibility of an executor to manage threads. When you send a task to an executor, it tries to use a pooled thread for the execution of this task, to avoid creating new threads. This mechanism is offered by the `Executor` interface and its implementing classes as the `ThreadPoolExecutor` class.

In this recipe, you're going to learn what information you can obtain about the status of a `ThreadPoolExecutor` executor and how to obtain it.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` that implements the `Runnable` interface.

```
public class Task implements Runnable {
```

2. Declare a private `long` attribute named `milliseconds`.

```
  private long milliseconds;
```

3. Implement the constructor of the class to initialize its attribute.

```
  public Task (long milliseconds) {
    this.milliseconds=milliseconds;
  }
```

4. Implement the `run()` method. Put the thread to sleep for the number of milliseconds specified by the `milliseconds` attribute.

```
  @Override
  public void run() {
```

```
    System.out.printf("%s: Begin\n",Thread.currentThread().
getName());
    try {
      TimeUnit.MILLISECONDS.sleep(milliseconds);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("%s: End\n",Thread.currentThread().
getName());

  }
```

5. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

6. Create a new `Executor` object using the `newCachedThreadPool()` method of the `Executors` class.

```
    ThreadPoolExecutor executor = (ThreadPoolExecutor)Executors.
newCachedThreadPool();
```

7. Create and submit 10 `Task` objects to the executor. Initialize the objects with a random number.

```
    Random random=new Random();
    for (int i=0; i<10; i++) {
      Task task=new Task(random.nextInt(10000));
      executor.submit(task);
    }
```

8. Create a loop with five steps. In each step, write information about the executor calling the `showLog()` method and put the thread to sleep for a second.

```
    for (int i=0; i<5; i++){
      showLog(executor);
      TimeUnit.SECONDS.sleep(1);
    }
```

9. Shut down the executor using the `shutdown()` method.

```
    executor.shutdown();
```

10. Create another loop with five steps In each step, write information about the executor calling the `showLog()` method and put the thread to sleep for a second.

```
for (int i=0; i<5; i++){
  showLog(executor);
  TimeUnit.SECONDS.sleep(1);
}
```

11. Wait for the finalization of the executor using the `awaitTermination()` method.

```
  executor.awaitTermination(1, TimeUnit.DAYS);
```

12. Display a message about the end of the program.

```
  System.out.printf("Main: End of the program.\n");
}
```

13. Implement the `showLog()` method that receives `Executor` as parameter. Write information about the size of the pool, the number of tasks, and the status of the executor.

```
  private static void showLog(ThreadPoolExecutor executor) {
    System.out.printf("*********************");
    System.out.printf("Main: Executor Log");
    System.out.printf("Main: Executor: Core Pool Size:
%d\n",executor.getCorePoolSize());
    System.out.printf("Main: Executor: Pool Size: %d\n",executor.
getPoolSize());
    System.out.printf("Main: Executor: Active Count:
%d\n",executor.getActiveCount());
    System.out.printf("Main: Executor: Task Count: %d\n",executor.
getTaskCount());
    System.out.printf("Main: Executor: Completed Task Count:
%d\n",executor.getCompletedTaskCount());
    System.out.printf("Main: Executor: Shutdown: %s\n",executor.
isShutdown());
    System.out.printf("Main: Executor: Terminating:
%s\n",executor.isTerminating());
    System.out.printf("Main: Executor: Terminated: %s\n",executor.
isTerminated());
    System.out.printf("*********************\n");
  }
```

## How it works...

In this recipe, you have implemented a task that blocks its execution thread for a random number of milliseconds. Then, you have sent 10 tasks to an executor and while you're waiting for their finalization, you have written information about the status of the executor to the console. You have used the following methods to get the status of the `Executor` object:

- `getCorePoolSize()`: This method returns an `int` number, which is the core number of threads. It's the minimum number of threads that will be in the internal thread pool when the executor is not executing any task.
- `getPoolSize()`: This method returns an `int` value, which is the actual size of the internal thread pool.
- `getActiveCount()`: This method returns an `int` number, which is the number of threads that are currently executing tasks.
- `getTaskCount()`: This method returns a `long` number, which is the number of tasks that have been scheduled for execution.
- `getCompletedTaskCount()`: This method returns a `long` number, which is the number of tasks that have been executed by this executor and have finished their execution.
- `isShutdown()`: This method returns a `Boolean` value when the `shutdown()` method of an executor has been called to finish its execution.
- `isTerminating()`: This method returns a `Boolean` value when the executor is doing the `shutdown()` operation, but it hasn't finished it yet.
- `isTerminated()`: This method returns a `Boolean` value when this executor has finished its execution.

## See also

- The *Creating a thread executor* recipe in *Chapter 4*, *Thread Executors*
- The *Customizing the ThreadPoolExecutor class* recipe in *Chapter 7*, *Customizing Concurrency Classes*
- The *Implementing a priority-based Executor class* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Monitoring a Fork/Join pool

The Executor framework provides a mechanism that allows the separation of the task implementation from the creation and management of the threads that execute those tasks. Java 7 includes an extension of the Executor framework for a specific kind of problem that will improve the performance of other solutions (as using `Thread` objects directly or the Executor framework). It's the Fork/Join framework.

This framework is designed to solve those problems that can be broken into smaller tasks using the divide and conquer technique using the `fork()` and `join()` operations. The main class that implements this behavior is the `ForkJoinPool` class.

In this recipe, you're going to learn what information you can obtain about a `ForkJoinPool` class and how to obtain it.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` that extends the `RecursiveAction` class.

   ```
   public class Task extends RecursiveAction{
   ```

2. Declare a private `int` array attribute named `array` to store the array of elements you want to increment.

   ```
   private int array[];
   ```

3. Declare two private `int` attributes named `start` and `end` to store the start and end positions of the block of elements this task has to process.

   ```
     private int start;
     private int end;
   ```

4. Implement the constructor of the class to initialize its attributes.

   ```
     public Task (int array[], int start, int end) {
       this.array=array;
       this.start=start;
       this.end=end;
     }
   ```

5. Implement the `compute()` method with the main logic of the task. If the task has to process more than 100 elements, divide that set of elements in two parts, create two tasks to execute those parts, start its execution with the `fork()` method, and wait for its finalization with the `join()` method.

   ```
     protected void compute() {
       if (end-start>100) {
         int mid=(start+end)/2;
         Task task1=new Task(array,start,mid);
         Task task2=new Task(array,mid,end);

         task1.fork();
         task2.fork();

         task1.join();
         task2.join();
   ```

6. If the task has to process 100 elements or less, increment those elements by putting the thread to sleep for 5 milliseconds after each operation.

```
    } else {
      for (int i=start; i<end; i++) {
        array[i]++;

        try {
          Thread.sleep(5);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

7. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

8. Create a `ForkJoinPool` object named `pool`.

```
    ForkJoinPool pool=new ForkJoinPool();
```

9. Create an array of integer numbers named `array` with 10,000 elements.

```
    int array[]=new int[10000];
```

10. Create a new `Task` object to process the whole array.

```
    Task task1=new Task(array,0,array.length);
```

11. Send the task for execution in the pool using the `execute()` method.

```
    pool.execute(task1);
```

12. While the task doesn't finish its execution, call the `showLog()` method to write information about the status of the `ForkJoinPool` class and put the thread to sleep for a second.

```
    while (!task1.isDone()) {
      showLog(pool);
      TimeUnit.SECONDS.sleep(1);
    }
```

13. Shut down the pool using the `shutdown()` method.

```
pool.shutdown();
```

14. Wait for the finalization of the pool using the `awaitTermination()` method.

```
pool.awaitTermination(1, TimeUnit.DAYS);
```

15. Call the `showLog()` method to write information about the status of the `ForkJoinPool` class and write a message in the console indicating the end of the program.

```
showLog(pool);
System.out.printf("Main: End of the program.\n");
```

16. Implement the `showLog()` method. It receives a `ForkJoinPool` object as a parameter and writes information about its status and the threads and tasks that are executing.

```
  private static void showLog(ForkJoinPool pool) {
    System.out.printf("**********************\n");
    System.out.printf("Main: Fork/Join Pool log\n");
    System.out.printf("Main: Fork/Join Pool: Parallelism:
%d\n",pool.getParallelism());
    System.out.printf("Main: Fork/Join Pool: Pool Size:
%d\n",pool.getPoolSize());
    System.out.printf("Main: Fork/Join Pool: Active Thread Count:
%d\n",pool.getActiveThreadCount());
    System.out.printf("Main: Fork/Join Pool: Running Thread Count:
%d\n",pool.getRunningThreadCount());
    System.out.printf("Main: Fork/Join Pool: Queued Submission:
%d\n",pool.getQueuedSubmissionCount());
    System.out.printf("Main: Fork/Join Pool: Queued Tasks:
%d\n",pool.getQueuedTaskCount());
    System.out.printf("Main: Fork/Join Pool: Queued Submissions:
%s\n",pool.hasQueuedSubmissions());
    System.out.printf("Main: Fork/Join Pool: Steal Count:
%d\n",pool.getStealCount());
    System.out.printf("Main: Fork/Join Pool: Terminated :
%s\n",pool.isTerminated());
    System.out.printf("**********************\n");
  }
```

## How it works...

In this recipe, you have implemented a task that increments elements of an array using a `ForkJoinPool` class and a `Task` class that extends the `RecursiveAction` class; one of the kind of tasks that you can execute in a `ForkJoinPool` class. While the tasks are processing the array, you print information about the status of the `ForkJoinPool` class to the console. You have used the following methods to get the status of the `ForkJoinPool` class:

- `getPoolSize()`: This method returns an `int` value, which is the number of worker threads of the internal pool of a fork join pool
- `getParallelism()`: This method returns the desired level of parallelism established for a pool
- `getActiveThreadCount()`: This method returns the number of threads that are currently executing tasks
- `getRunningThreadCount()`: This method returns the number of working threads that are not blocked in any synchronization mechanism
- `getQueuedSubmissionCount()`: This method returns the number of tasks that have been submitted to a pool that haven't started their execution yet
- `getQueuedTaskCount()`: This method returns the number of tasks that have been submitted to a pool that have started their execution
- `hasQueuedSubmissions()`: This method returns a `Boolean` value indicating if this pool has queued tasks that haven't started their execution yet
- `getStealCount()`: This method returns a `long` value with the number of times a worker thread has stolen a task from another thread
- `isTerminated()`: This method returns a `Boolean` value indicating if the fork/join pool has finished its execution

## See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*
- The *Implementing the ThreadFactory interface to generate custom threads for the Fork/Join framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*
- The *Customizing tasks running in the Fork/Join framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*

# Writing effective log messages

A **log system** is a mechanism that allows you to write information to one or more destinations. A **Logger** has the following components:

- **One or more handlers**: A handler will determine the destination and the format of log messages. You can write log messages to the console, a file, or a database.
- **A name**: Usually, the name of a Logger used in a class that's based on the class name and its package name.
- **A level**: Log messages have a level associated that indicates its importance. A Logger also has a level used to decide what messages it is going to write. It only writes the messages that are as important as, or more important, than its level.

You should use the log system with the following two main purposes:

- Write as much information as you can when an exception is caught. This will help to localize the error and resolve it.
- Write information about the classes and methods that the program is executing.

In this recipe, you will learn how to use the classes provided by the `java.util.logging` package to add a log system to your concurrent application.

## Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `MyFormatter` that extends the `java.util.logging.Formatter` class. Implement the abstract `format()` method. It receives a `LogRecord` object as a parameter and returns a `String` object with the log message.

```
public class MyFormatter extends Formatter {
  @Override
  public String format(LogRecord record) {

    StringBuilder sb=new StringBuilder();
    sb.append("["+record.getLevel()+"] - ");
    sb.append(new Date(record.getMillis())+" : ");
      sb.append(record.getSourceClassName()+ "."+record.
```

```
getSourceMethodName()+" : ");
    sb.append(record.getMessage()+"\n");.
    return sb.toString();
  }
```

2. Create a class named `MyLogger`.

```
public class MyLogger {
```

3. Declare a private static `Handler` attribute named `handler`.

```
  private static Handler handler;
```

4. Implement the public static method `getLogger()` to create the `Logger` object that you're going to use to write the log messages. It receives a `String` parameter named `name`.

```
  public static Logger getLogger(String name){
```

5. Get `java.util.logging.Logger` associated with the name received as a parameter using the `getLogger()` method of the `Logger` class.

```
    Logger logger=Logger.getLogger(name);
```

6. Establish the log level to write all the log messages using the `setLevel()` method.

```
    logger.setLevel(Level.ALL);
```

7. If the handler attribute has the `null` value, create a new `FileHandler` object to write the log messages in the `recipe8.log` file. Assign to that handler a `MyFormatter` object as a formatter using the `setFormatter()` object.

```
    try {
      if (handler==null) {
        handler=new FileHandler("recipe8.log");
        Formatter format=new MyFormatter();
        handler.setFormatter(format);
      }
```

8. If the `Logger` object does not have a handler associated to it, assign the handler using the `addHandler()` method.

```
      if (logger.getHandlers().length==0) {
        logger.addHandler(handler);
      }
    } catch (SecurityException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
```

9. Return the `Logger` object created.

```
    return logger;
  }
```

10. Create a class named `Task` that implements the `Runnable` interface. It will be the task used to test your `Logger` object.

```
public class Task implements Runnable {
```

11. Implement the `run()` method.

```
  @Override
  public void run() {
```

12. First, declare a `Logger` object named `logger`. Initialize it using the `getLogger()` method of the `MyLogger` class passing the name of this class as a parameter.

```
    Logger logger= MyLogger.getLogger(this.getClass().getName());
```

13. Write a log message indicating the beginning of the execution of the method using the `entering()` method.

```
    logger.entering(Thread.currentThread().getName(), "run()");
Sleep the thread for two seconds.
    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

14. Write a log message indicating the end of the execution of the method using the `exiting()` method.

```
    logger.exiting(Thread.currentThread().getName(),
"run()",Thread.currentThread());
  }
```

15. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

16. Declare a `Logger` object named `logger`. Initialize it using the `getLogger()` method of the `MyLogger` class passing the string `Core` as a parameter.

```
    Logger logger=MyLogger.getLogger("Core");
```

17. Write a log message indicating the start of the execution of the main program using the `entering()` method.

```
logger.entering("Core", "main()",args);
```

18. Create a `Thread` array to store five threads.

```
Thread threads[]=new Thread[5];
```

19. Create five `Task` objects and five threads to execute them. Write log messages to indicate that you're going to launch a new thread and to indicate that you have created the thread.

```
    for (int i=0; i<threads.length; i++) {
      logger.log(Level.INFO,"Launching thread: "+i);
      Task task=new Task();
      threads[i]=new Thread(task);
      logger.log(Level.INFO,"Thread created: "+ threads[i].
getName());
      threads[i].start();
    }
```

20. Write a log message to indicate that you have created the threads.

```
    logger.log(Level.INFO,"Ten Threads created."+
"Waiting for its finalization");
```

21. Wait for the finalization of the five threads using the `join()` method. After the finalization of each thread, write a log message indicating that the thread has finished.

```
    for (int i=0; i<threads.length; i++) {
      try {
        threads[i].join();
        logger.log(Level.INFO,"Thread has finished its
execution",threads[i]);
      } catch (InterruptedException e) {
        logger.log(Level.SEVERE, "Exception", e);
      }
    }
```

22. Write a log message to indicate the end of the execution of the main program using the `exiting()` method.

```
    logger.exiting("Core", "main()");
  }
```

## How it works...

In this recipe, you have used the `Logger` class provided for the Java logging API to write log messages in a concurrent application. First of all, you have implemented the `MyFormatter` class to give a format to the log messages. This class extends the `Formatter` class that declares the abstract method `format()`. This method receives a `LogRecord` object with all the information of the log message and returns a formatted log message. In your class, you have used the following methods of the `LogRecord` class to obtain information about the log message:

- `getLevel()`: Returns the level of a message
- `getMillis()`: Returns the date when a message was sent to a `Logger` object
- `getSourceClassName()`: Returns the name of a class that sent the message to the Logger
- `getSourceMessageName()`: Returns the name of a method that sent the message to the Logger

`getMessage()` returns the log message. The `MyLogger` class implements the static method `getLogger()` that creates a `Logger` object and assigns a `Handler` object to write log messages of the application to the `recipe8.log` file using the `MyFormatter` formatter. You create the `Logger` object with the static method `getLogger()` of that class. This method returns a different object per name that is passed as a parameter. You only have created one `Handler` object, so all the `Logger` objects will write its log messages in the same file. You also have configured the logger to write all the log messages, regardless of its level.

Finally, you have implemented a `Task` object and a main program that writes different log messages in the logfile. You have used the following methods:

- `entering()`: Write a message with the `FINER` level indicating that a method starts its execution
- `exiting()`: Write a message with the `FINER` level indicating that a method ends its execution
- `log()`: Write a message with the specified level

## There's more...

When you work with a log system, you have to take into consideration two important points:

- **Write the necessary information**: If you write too little information, the logger won't be useful because it won't fulfil its purpose. If you write too much information, you will generate too large logfiles that will be unmanageable and make it difficult to get the necessary information.

- **Use the adequate level for the messages**: If you write information messages with the higher level or error messages with a lower level, you will confuse the user who looks at the logfiles. It will be more difficult to know what happened in an error situation or you will have too much information to know the main cause of the error.

There are other libraries that provide a log system more complete than the `java.util.logging` package, such as the Log4j or slf4j libraries. But the `java.util.logging` package is part of the Java API and all its methods are multi-thread safe, so we can use it in concurrent applications without problems.

## See also

- The *Using non-blocking thread-safe lists* recipe in *Chapter 6*, *Concurrent Collections*
- The *Using blocking thread-safe lists* recipe in *Chapter 6*, *Concurrent Collections*
- The *Using blocking thread-safe lists ordered by priority* recipe in *Chapter 6*, *Concurrent Collections*
- The *Using thread-safe lists with delayed elements* recipe in *Chapter 6*, *Concurrent Collections*
- The *Using thread-safe navigable maps* recipe in *Chapter 6*, *Concurrent Collections*
- The *Generating concurrent random numbers* recipe in *Chapter 6*, *Concurrent Collections*

# Analyzing concurrent code with FindBugs

The **static code analysis tools** are a set of tools that analyze the source code of an application looking for potential errors. These tools, such as Checkstyle, PMD, or FindBugs have a set of predefined rules of good practices and parse the source code looking for violations of those rules. The objective is to find errors or places causing poor performance early, before it will be executed in production. Programming languages usually offer such tools and Java is not an exception. One of these tools to analyze Java code is FindBugs. It's an open source tool that includes a series of rules to analyze Java-concurrent code.

In this recipe, you will learn how to use this tool to analyze your Java-concurrent applications.

## Getting ready

Before stating this recipe, you should download FindBugs from the project's web page (`http://findbugs.sourceforge.net/`). You can download a standalone application or an Eclipse plugin. In this recipe, you will use the standalone version.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` that extends the `Runnable` interface.

```
public class Task implements Runnable {
```

2. Declare a private `ReentrantLock` attribute named `Lock`.

```
  private ReentrantLock lock;
```

3. Implement a constructor of the class.

```
  public Task(ReentrantLock lock) {
    this.lock=lock;
  }
```

4. Implement the `run()` method. Get the control of the lock, put the thread to sleep for 2 seconds and free the lock.

```
  @Override
  public void run() {
    lock.lock();
    try {
      TimeUnit.SECONDS.sleep(1);
      lock.unlock();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
```

5. Create the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

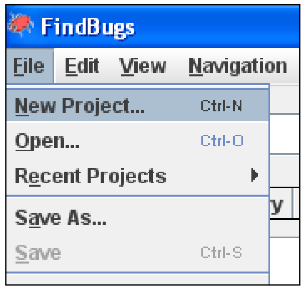
6. Declare and create a `ReentrantLock` object named `lock`.

```
    ReentrantLock lock=new ReentrantLock();
```

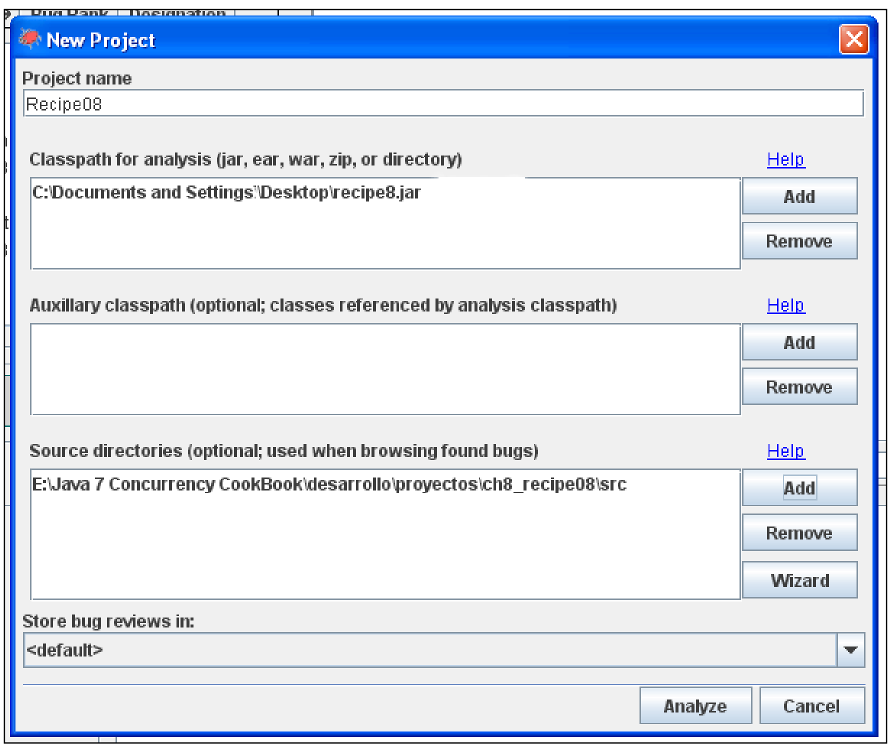
7. Create 10 `Task` objects and 10 threads to execute those tasks. Start the threads calling the `run()` method.

```
    for (int i=0; i<10; i++) {
      Task task=new Task(lock);
      Thread thread=new Thread(task);
      thread.run();
    }
  }
```

8. Export the project as a `jar` file. Call it `recipe8.jar`. Use the menu option of your IDE or the `javac` and `jar` commands to compile and compress your application.
9. Start the FindBugs standalone application running the `findbugs.bat` command in Windows or the `findbugs.sh` command in Linux.
10. Create a new project with the **New Project** option of the **File** menu in the menu bar.



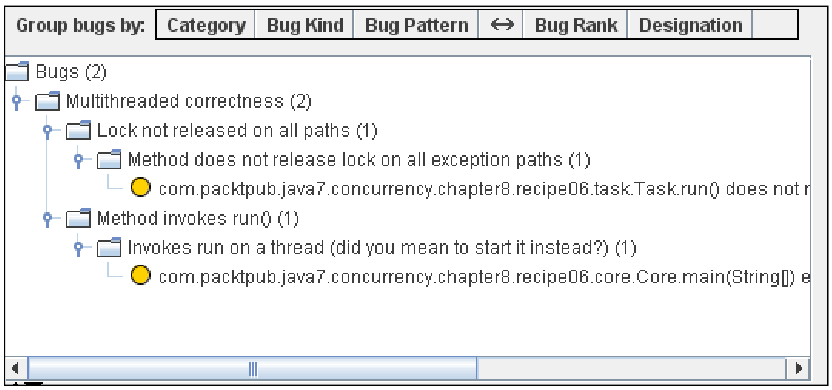

11. The **FindBugs** application shows a window to configure the project. In the **Project Name** field introduce the text `Recipe08`. In the **Classpath for analysis** field add the `jar` file with the project and in the **Source directories** field add the directory with the source code of the example. Refer to the following screenshot:



12. Click on the **Analyze** button to create the new project and analyze its code.
13. The **FindBugs** application shows the result of the analysis of the code. In this case, it has found two bugs.
14. Click one of the bugs and you'll see the source code of the bug in the right-hand side panel and the description of the bug in the panel of the bottom of the screen.

## How it works...

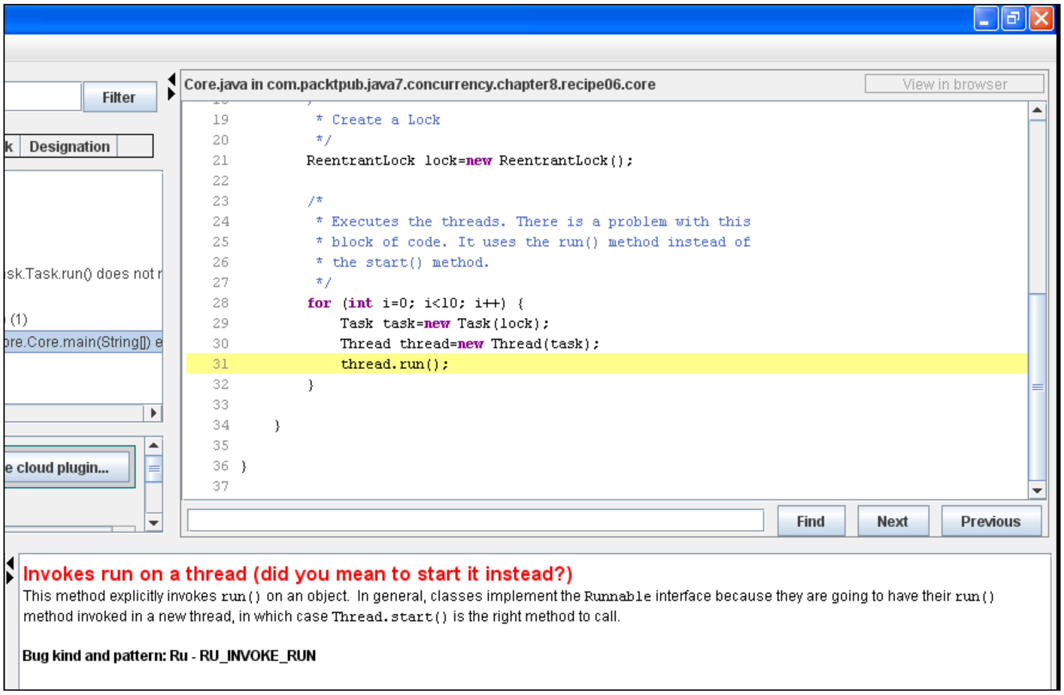
The following screenshot shows the result of the analysis by FindBugs:




The analysis has detected the following two potential bugs in the application:

- One in the `run()` method of the class `Task`. If an `InterruptedException` exception is thrown, the task doesn't free the lock because it won't execute the `unlock()` method. This will probably cause a deadlock situation in the application.
- The other is in the `main()` method of the `Main` class because you have called the `run()` method of a thread directly , but not the `start()` method to begin the execution of the thread.

If you make a double-click in one of the two bugs, you will see detailed information about it. As you have included the source-code reference in the configuration of the project, you also will see the source code where the bug was detected. The following screenshot shows you an example of this:




## There's more...

Be aware that FindBugs can only detect some problematic situations (related or not with the concurrency code). For example, if you delete the `unlock()` call in the `run()` method of the `Task` class and repeat the analysis, FindBugs won't alert you that you get the lock in the task but you never free it.

Use the tools for the static code analysis as a help to improve the quality of your code, but do not expect to detect all the bugs in your code.

## See also

- The *Configuring NetBeans for debugging concurrency code* recipe in *Chapter 8, Testing Concurrency Applications*

# Configuring Eclipse for debugging concurrency code

Nowadays, almost every programmer, regardless of the programming language in use, create their applications with an IDE. They provide lots of interesting functionalities integrated in the same application, such as:

- Project management
- Automatic code generation
- Automatic documentation generation
- Integration with control version systems
- A debugger to test the applications
- Different wizards to create projects and elements of the applications

One of the most helpful features of an IDE is a debugger. You can execute your application step-by-step and analyze the values of all the objects and variables of your program.

If you work with the Java programming language, Eclipse is one of the most popular IDEs. It has an integrated debugger that allows you to test your applications. By default, when you debug a concurrent application and the debugger finds a breakpoint, it only stops the thread that has that breakpoint while the rest of the threads continue with their execution.

In this recipe, you will learn how to change that configuration to help you to test concurrent applications.

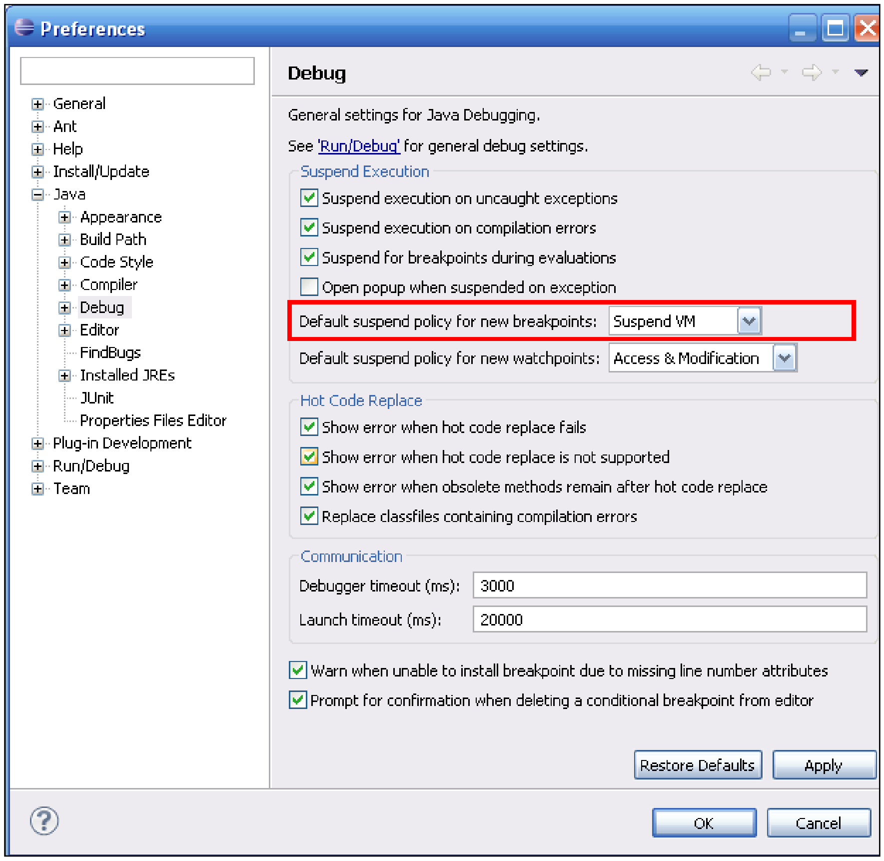
## Getting ready

You must have installed the Eclipse IDE. Open it and select a project with a concurrent application implemented in it, for example, one of the recipes implemented in the book.

## How to do it...

Follow these steps to implement the example:

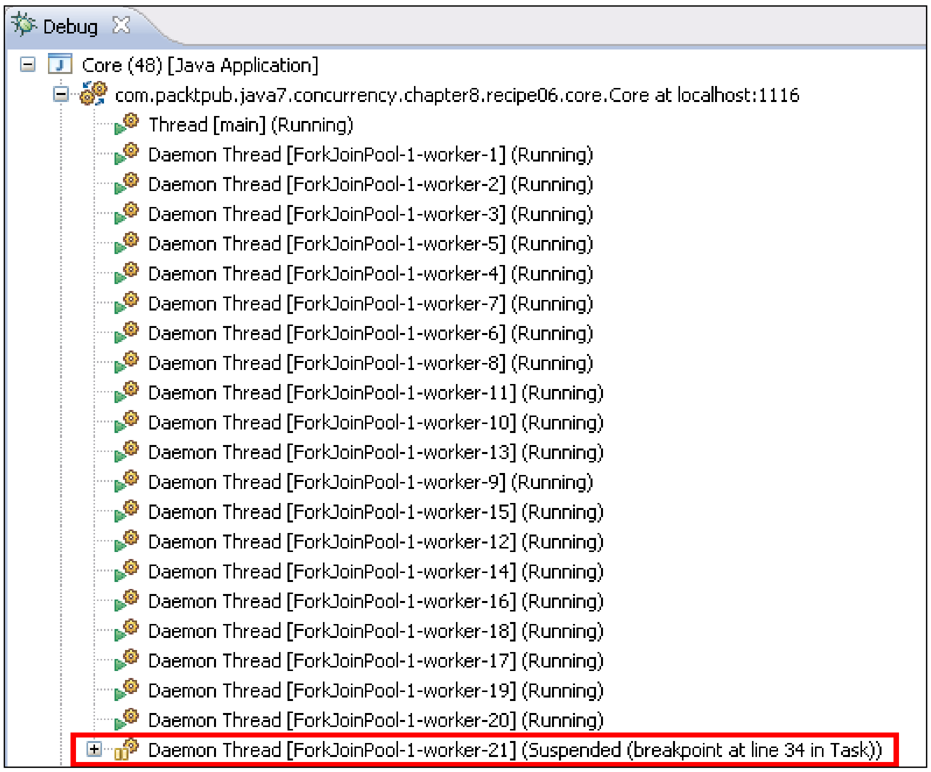
1. Select the menu option **Window** | **Preferences**.
2. In the left-hand side menu, expand the **Java** option.
3. In the left-hand side menu, select the **Debug** option. The following screenshot shows the appearance of that window:



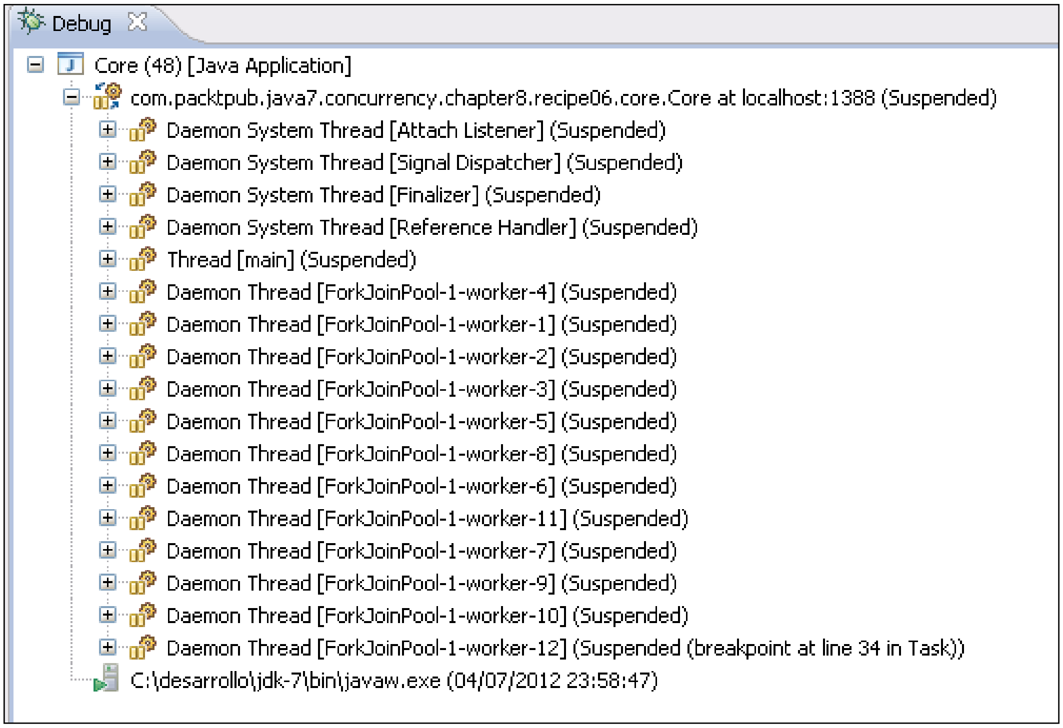

4. Change the value of the **Default suspend policy for new breakpoints** from **Suspend Thread** to **Suspend VM** (marked in red in the screenshot).
5. Click on the **OK** button to confirm the change.

## How it works...

As we mentioned in the introduction of this recipe, by default, when you debug a concurrent Java application in Eclipse and the debug process finds a breakpoint, it only suspends the thread that hit the breakpoint first while the other threads continue with their execution. The following screenshot shows an example of that situation:




You can see that only the **worker-21** is suspended (marked in red in the screenshot) while the rest of the threads are running. However, if you change **Default suspend policy for new breakpoints** to **Suspend VM**, all the threads suspend their execution while you're debugging a concurrent application and the debug process hits a breakpoint. The following screenshot shows an example of this situation:




With the change, you can see that all the threads are suspended. You can continue debugging any thread you want. Choose the suspend policy that best suits your needs.

# Configuring NetBeans for debugging concurrency code

In today's world, software is necessary to develop applications that work properly, that meet the quality standards of the company, and that will be easily modified in the future, in a limited time and with a cost as low as possible. To achieve this goal, it is essential to use an IDE that integrates under one common interface several tools (compilers and debuggers) that facilitate the development of applications.

If you work with the Java programming language, NetBeans is one of the most popular IDEs. It has an integrated debugger that allows you to test your application.

In this recipe, you will learn how to change that configuration to help you to test concurrent applications.

## Getting ready

You should have the NetBeans IDE installed. Open it and create a new Java project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `Task1` and specify that it implements the `Runnable` interface.

```
public class Task1 implements Runnable {
```

2. Declare two private `Lock` attributes named `lock1` and `lock2`.

```
    private Lock lock1, lock2;
```

3. Implement the constructor of the class to initialize its attributes.

```
    public Task1 (Lock lock1, Lock lock2) {
        this.lock1=lock1;
        this.lock2=lock2;
    }
```

4. Implement the `run()` method. First, get the control of the `lock1` object using the `lock()` method and write a message in the console indicating that you have got it.

```
 @Override
    public void run() {
        lock1.lock();
        System.out.printf("Task 1: Lock 1 locked\n");
```

5. Then, get the control of the `lock2` object using the `lock()` method and write a message in the console indicating that you have got it.

```
        lock2.lock();
        System.out.printf("Task 1: Lock 2 locked\n");
Finally, release the two lock objects. First, the lock2 object and
then the lock1 object.
        lock2.unlock();
        lock1.unlock();
    }
```

6. Create a class named `Task2` and specify that it implements the `Runnable` interface.

```
public class Task2 implements Runnable{
```

7. Declare two private `Lock` attributes named `lock1` and `lock2`.

```
    private Lock lock1, lock2;
```

8. Implement the constructor of the class to initialize its attributes.

```
    public Task2(Lock lock1, Lock lock2) {
        this.lock1=lock1;
        this.lock2=lock2;
    }
```

9. Implement the `run()` method. First, get the control of the `lock2` object using the `lock()` method and write a message in the console indicating that you have got it.

```
 @Override
    public void run() {
        lock2.lock();
        System.out.printf("Task 2: Lock 2 locked\n");
```

10. Then, get the control of the `lock1` object using the `lock()` method and write a message in the console indicating that you have got it.

```
        lock1.lock();
        System.out.printf("Task 2: Lock 1 locked\n");
```

11. Finally, release the two lock objects. First, the `lock1` object and then the `lock2` object.

```
        lock1.unlock();
        lock2.unlock();
    }
```

12. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
```

13. Create two lock objects named `lock1` and `lock2`.

```
        Lock lock1, lock2;
        lock1=new ReentrantLock();
        lock2=new ReentrantLock();
```

14. Create a `Task1` object named `task1`.

```
        Task1 task1=new Task1(lock1, lock2);
```

15. Create a `Task2` object named `task2`.

```
        Task2 task2=new Task2(lock1, lock2);
```

16. Execute both tasks using two threads.

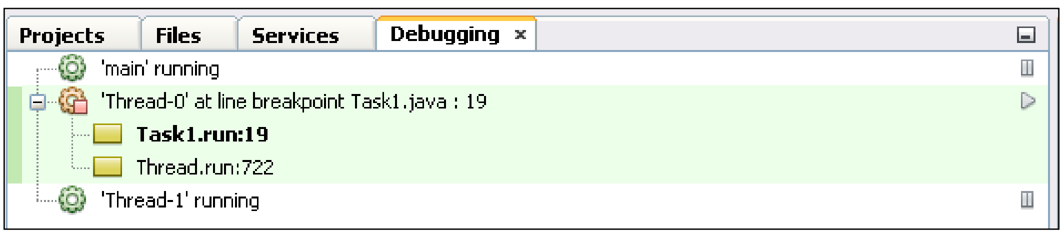
```
        Thread thread1=new Thread(task1);
        Thread thread2=new Thread(task2);

        thread1.start();
        thread2.start();
```

17. While the two tasks haven't finished their execution, write a message in the console every 500 milliseconds. Use the `isAlive()` method to check if a thread has finished its execution.

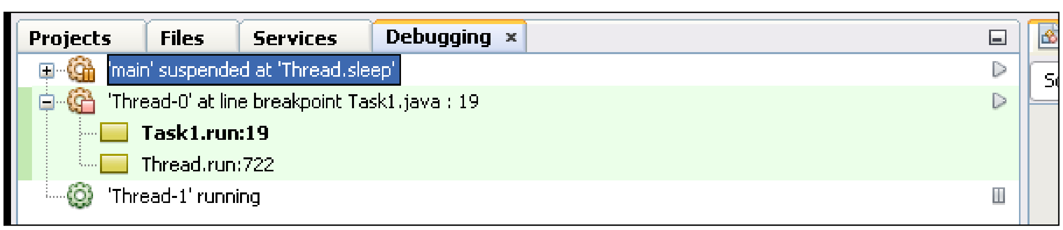
```
while ((thread1.isAlive()) &&(thread2.isAlive())) {
    System.out.println("Main: The example is"+ "running");
    try {
        TimeUnit.MILLISECONDS.sleep(500);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

18. Add a breakpoint in the first call to the `println()` method of the `run()` method of the `Task1` class.

19. Debug the program. You will see the **Debugging** window in the top left-hand side corner of the main NetBeans window. The next screenshot presents the appearance of that window with the thread that executes the `Task1` object slept because they have arrived at the breakpoint and the other threads running:




20. Pause the execution of the main thread. Select that thread, right-click, and select the **Suspend** option. The following screenshot shows the new appearance of the **Debugging** window. Refer to the following screenshot:




21. Resume the two paused threads. Select each thread, right-click, and select the **Resume** option.

## How it works...

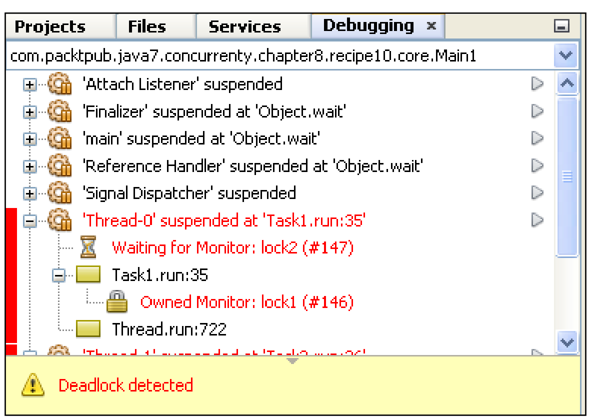
While debugging a concurrent application using NetBeans, when the debugger hits a breakpoint, it suspends the thread that hit the breakpoint and shows the **Debugging** window in the top left-hand side corner with the threads that are currently running.

You can use the window to pause or resume the threads that are currently running using the **Pause** or **Resume** options. You can also see the values of the variables or attributes of the threads using the **Variables** tab.

NetBeans also includes a deadlock detector. When you select the **Check for Deadlock** option in the **Debug** menu, NetBeans performs an analysis of the application that you're debugging to determine if there's a deadlock situation. This example presents a clear deadlock. The first thread gets the lock `lock1` first and then the lock `lock2`. The second thread gets the locks just in a reverse manner. The breakpoint inserted provokes the deadlock, but if you use NetBeans deadlock detector, you'll not find anything, so this option should be used with caution. Change the locks used in both tasks by the `synchronized` keyword and debug the program again. The code of the `Task1` will be presented as follows:

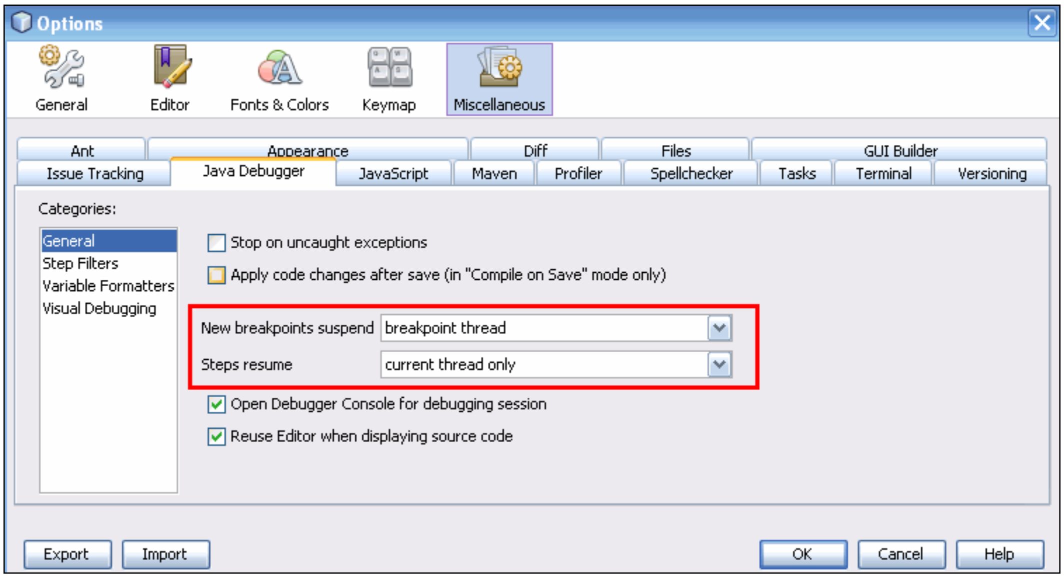
```
@Override
public void run() {
    synchronized(lock1) {
        System.out.printf("Task 1: Lock 1 locked\n");
        synchronized(lock2) {
            System.out.printf("Task 1: Lock 2 locked\n");
        }
    }
}
```

The code of the `Task2` class will be analogous to this, but changes the order of the locks. If you debug the example again, you will obtain a deadlock again, but in this case, it's detected by the deadlock detector, as you can see in the following screenshot:



## There's more...

There are options to control the debugger. Select the **Options** option in the **Tools** menu. Then, select the **Miscellaneous** option and the **Java Debugger** tab. The following screenshot shows the appearance of that window:




There are two options on that window that control the behavior described earlier:

- **New breakpoints suspend**: With this option, you can configure the behavior of NetBeans, which finds a breakpoint in a thread. You can suspend only the thread that has the breakpoint or all the threads of the application.
- **Steps resume**: With this option, you can configure the behavior of NetBeans when you resume a thread. You can resume only the current thread or all the threads.

Both options have been marked in the screenshot presented earlier.

## See also

- The *Configuring Eclipse for debugging concurrency code* recipe in *Chapter 8*, *Testing Concurrent Applications*

# Testing concurrency code with MultithreadedTC

MultithreadedTC is a Java library for testing concurrent applications. Its main objective is to solve the problem of concurrent applications being non-deterministic. You can't control their order of execution. For this purpose, it includes an internal **metronome** to control the order of execution of the different threads that form the application. Those testing threads are implemented as methods of a class.

In this recipe, you will learn how to use the MultithreadedTC library to implement a test for `LinkedTransferQueue`.

## Getting ready

You must also download the MultithreadedTC library from `http://code.google.com/p/multithreadedtc/` and the JUnit library, Version 4.10 from `http://www.junit.org/`. Add the files `junit-4.10.jar` and `MultithreadedTC-1.01.jar` to the libraries of the project.

## How to do it...

Follow these steps to implement the example:

1. Create a class named `ProducerConsumerTest` that extends the `MultithreadedTestCase` class.

   ```
   public class ProducerConsumerTest extends MultithreadedTestCase {
   ```

2. Declare a private `LinkedTransferQueue` attribute parameterized with the `String` class named `queue`.

   ```
     private LinkedTransferQueue<String> queue;
   ```

3. Implement the `initialize()` method. This method won't receive any parameters and returns no value. It calls the `initialize()` method of its parent class and then initializes the queue attribute.

   ```
     @Override
     public void initialize() {
       super.initialize();
       queue=new LinkedTransferQueue<String>();
       System.out.printf("Test: The test has been initialized\n");
     }
   ```

4. Implement the `thread1()` method. It will implement the logic of the first consumer. Call the `take()` method of the queue and then write the returned value in the console.

```
public void thread1() throws InterruptedException {
  String ret=queue.take();
  System.out.printf("Thread 1: %s\n",ret);
}
```

5. Implement the `thread2()` method. It will implement the logic of the second consumer. First, wait until the first thread has slept in the `take()` method using the `waitForTick()` method. Then, call the `take()` method of the queue and then write the returned value in the console.

```
public void thread2() throws InterruptedException {
  waitForTick(1);
  String ret=queue.take();
  System.out.printf("Thread 2: %s\n",ret);
}
```

6. Implement the `thread3()` method. It will implement the logic of a producer. First, wait until the two consumers are blocked in the `take()` method using the `waitForTick()` method twice. Then, call the `put()` method of the queue to insert two `Strings` in the queue.

```
public void thread3() {
  waitForTick(1);
  waitForTick(2);
  queue.put("Event 1");
  queue.put("Event 2");
  System.out.printf("Thread 3: Inserted two elements\n");
}
```

7. Finally, implement the `finish()` method. Write a message in the console to indicate that the test has finished its execution. Check that the two events have been consumed (so the size of the queue is `0`) using the `assertEquals()` method.

```
public void finish() {
  super.finish();
  System.out.printf("Test: End\n");
  assertEquals(true, queue.size()==0);
  System.out.printf("Test: Result: The queue is empty\n");
}
```

8. Implement the main class of the example by creating a class named `Main` with a `main()` method.

```
public class Main {

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

9. Create a `ProducerConsumerTest` object named `test`.

```
    ProducerConsumerTest test=new ProducerConsumerTest();
```

10. Execute the test using the `runOnce()` method of the `TestFramework` class.

```
     System.out.printf("Main: Starting the test\n");
    TestFramework.runOnce(test);
    System.out.printf("Main: The test has finished\n");
```

## How it works...

In this recipe, you have implemented a test for the `LinkedTransferQueue` class using the MultithreadedTC library. You can implement a test to any concurrent application or class using this library and its metronome. In the example, you have implemented the classical producer/consumer problem with two consumers and a producer. You want to test that the first `String` object introduced in the buffer is consumed by the first consumer that arrives at the buffer and the second `String` object introduced in the buffer is consumed by the second consumer that arrives at the buffer.

The MultithreadedTC library is based on the JUnit library, which is the most often used library to implement unit tests in Java. To implement a basic test using the MultithreadedTC library, you have to extend the `MultithreadedTestCase` class. This class extends the `junit.framework.AssertJUnit` class that includes all the methods to check the results of the test. It doesn't extend the `junit.framework.TestCase` class, so you can't integrate the MultithreadedTC tests with other JUnit tests.

Then, you can implement the following methods:

- `initialize()`: The implementation of this method is optional. It's executed when you start the test, so you can use it to initialize objects that are using the test.
- `finish()`: The implementation of this method is optional. It's executed when the test has finished. You can use it to close or release resources used during the test or to check the results of the test.
- Methods that implement the test: These methods have the main logic of the test you implement. They have to start with the `thread` keyword followed by a string. For example, `thread1()`.

To control the order of execution of threads, you use the `waitForTick()` method. This method receives an `integer` type as a parameter and puts the thread that is executing the method to sleep until all threads that are running in the test are blocked. When they are blocked, the MultithreadedTC library resumes the threads that are blocked by a call to the `waitForTick()` method.

The integer you pass as a parameter of the `waitForTick()` method is used to control the order of execution. The metronome of the MultithreadedTC library has an internal counter. When all the threads are blocked, the library increments that counter to the next number specified in the `waitForTick()` calls that are blocked.

Internally, when the MultithreadedTC library has to execute a test, first it executes the `initialize()` method. Then, it creates a thread per method that starts with the `thread` keyword (in your example, the methods `thread1()`, `thread2()`, and `thread3()`) and when all the threads have finished their execution, it executes the `finish()` method. To execute the test, you have used the `runOnce()` method of the `TestFramework` class.

## There's more...

If the MultithreadedTC library detects that all the threads of the test are blocked, but none of them are blocked in the `waitForTick()` method, the test is declared to be in a deadlock state and a `java.lang.IllegalStateException` exception is thrown.

## See also

- The *Analyzing concurrent code with FindBugs* recipe in *Chapter 8, Testing Concurrent Applications*

# Index

## D



## E



## F

## I



## J



## K



## L

# T

## U



## V



## W



## Y

## Thank you for buying
## Java 7 Concurrency Cookbook

# About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

# About Packt Enterprise

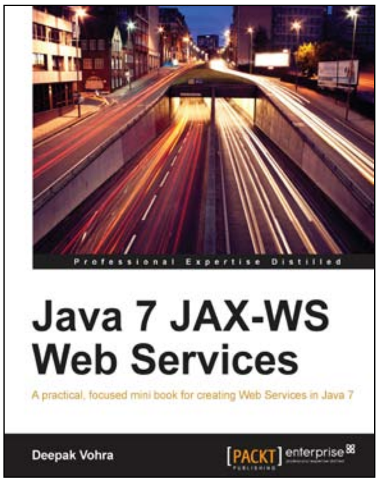
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

# Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING



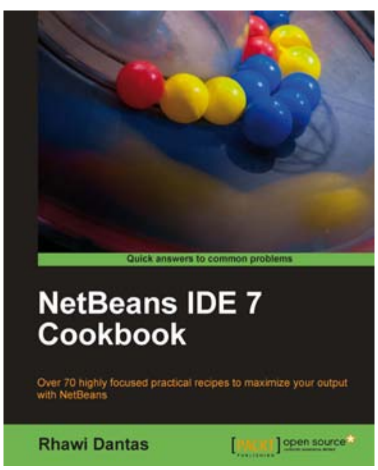

## Java 7 JAX-WS Web Services

ISBN: 978-1-84968-720-1 Paperback: 64 pages

A practical, focused mini book for creating Web Services in Java 7

1. Develop Java 7 JAX-WS web services using the NetBeans IDE and Oracle GlassFish server
2. End-to-end application which makes use of the new clientjar option in JAX-WS wsimport tool
3. Packed with ample screenshots and practical instructions

---




## NetBeans IDE 7 Cookbook

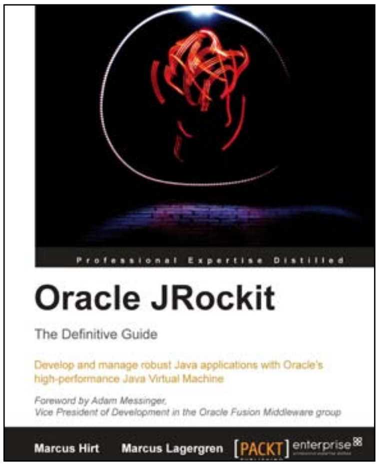
ISBN: 978-1-84951-250-3 Paperback: 308 pages

Over 70 highly focused practical recipes to maximize your output with NetBeans

1. Covers the full spectrum of features offered by the NetBeans IDE
2. Discover ready-to-implement solutions for developing desktop and web applications
3. Learn how to deploy, debug, and test your software using NetBeans IDE
4. Another title in Packt's Cookbook series giving clear, real-world solutions to common practical problems

Please check **www.PacktPub.com** for information on our titles

[PACKT] enterprise
professional expertise distilled
PUBLISHING




## Oracle JRockit: The Definitive Guide

ISBN: 978-1-84719-806-8 Paperback: 588 pages

Develop and manage robust Java applications with Oracle's high-performance Java Virtual Machine

1. Learn about the fundamental building blocks of a JVM, such as code generation and memory management, and utilize this knowledge to develop code you can count on
2. Realize the full potential of Java applications by learning how to apply advanced tuning and analysis
3. Work with the JRockit Mission Control 3.1/4.0 tools suite to debug or profile your Java applications

---


## Java 7 New Features Cookbook

ISBN: 978-1-84968-562-7 Paperback: 384 pages

Over 100 comprehensive recipes to get you up-to-speed with all the exciting new features of Java 7

1. Comprehensive coverage of the new features of Java 7 organized around easy-to-follow recipes
2. Covers exciting features such as the try-with-resources block, the monitoring of directory events, asynchronous IO and new GUI enhancements, and more
3. A learn-by-example based approach that focuses on key concepts to provide the foundation to solve real world problems

Please check **www.PacktPub.com** for information on our titles