9
Additional Information

In this chapter, we will cover the following topics:

- Synchronizing a block of code
- Processing results for `Runnable` objects in the `Executor` framework
- Processing uncontrolled exceptions in a `ForkJoinPool` class
- Using blocking thread-safe lists for communicating with producers and consumers
- Monitoring a `Thread` class
- Monitoring a `Semaphore` class

Introduction

The recipes in this chapter include notions of synchronization, the `Executor` framework and the `Fork/Join` framework, concurrent data structures, and monitoring of concurrent objects.

Synchronizing a block of code

In the *Synchronizing a method* recipe of *Chapter 2*, *Basic Thread Synchronization*, you can learn how to use the `synchronized` keyword in the declaration of a method to guarantee that only one execution thread has concurrent access to that method (and to all synchronized methods) of an object of that class.

Using the `synchronized` keyword in this way may have some disadvantages. If you have very long operations in the synchronized methods and those long operations do not work with the shared data, there will be execution threads blocked waiting for the completion of those operations when they could be running. Examples of those kind of operations are database operations or disk operations.

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 that the critical section (the block of code that can be accessed only by one thread at a time) be as short as possible.

In this recipe, you will learn how to use the `synchronized` keyword to protect a block of code developing an example of the situation explained before.

Getting ready

You should read the *Synchronizing a method* recipe in *Chapter 2*, *Basic Thread Synchronization* to obtain a better understanding of this recipe.

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

How to do it...

Perform the following steps to implement the example:

1. Create a class called `BuildStats` that stores the number of people inside a building. It has only one long attribute called `numPeople`.

```
public class BuildStats {

  private long numPeople;
```

2. Implement the `comeIn()` method. This method increments the number of people inside the building. It uses the `synchronized` keyword to protect the access to shared data, but leaves out of the synchronized block the rest of the instructions of the method.

```
  public /*synchronized*/ void comeIn(){
    System.out.printf("%s: A person enters.\n", Thread.
currentThread().getName());
    synchronized (this) {
      numPeople++;
    }
    generateCard();
  }
```

3. Implement the `goOut()` method. This method decrements the number of people inside the building. It uses the `synchronized` keyword to protect the access to shared data, but leaves out of the synchronized block the rest of the instructions of the method.

```
  public /*synchronized*/ void goOut(){
    System.out.printf("%s: A person leaves.\n", Thread.
currentThread().getName());
    synchronized (this) {
      numPeople--;
    }
    generateReport();
  }
```

4. Implement the auxiliary methods `generateCard()`, `generateReport()`, and `printStats()`.

```
  private void generateCard(){
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  private void generateReport(){
    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  public void printStats(){
    System.out.printf("%d persons in the building.\n", numPeople);
}
```

5. Implement the `Sensor1` class. This class simulates a sensor in a door of the building that controls people coming in and leaving the building. It must implement the `Runnable` interface to run this task as a thread.

```
public class Sensor1 implements Runnable {
```

6. Declare in this class a `BuildStats` object and implement the constructor of the class that initializes that object.

```
  private BuildStats stats;

  public Sensor1(BuildStats stats){
    this.stats=stats;
  }
```

7. Implement the `run()` method. It simulates the entry and exit of several people.

```
@Override
 public void run() {
  stats.comeIn();
  stats.comeIn();
  stats.comeIn();
  stats.goOut();
  stats.comeIn();
}
```

8. Implement the `Sensor2` class. This class simulates a sensor in a door of the building that controls people coming in and leaving the building. It must implement the `Runnable` interface to run this task as a thread.

```
public class Sensor2 implements Runnable {
```

9. Declare in this class a `BuildStats` object and implement the constructor of the class that initializes that object.

```
private BuildStats stats;

public Sensor2(BuildStats stats){
  this.stats=stats;
}
```

10. Implement the `run()` method. It simulates the entry and exit of several people.

```
 @Override
public void run() {
  stats.comeIn();
  stats.comeIn();
  stats.goOut();
  stats.goOut();
  stats.goOut();
}
```

11. Implement the main class of the application. Implement a class called `Main` and add to it the `main()` method.

```
public class Main {

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

12. Create a `BuildStats` object.

```
    BuildStats stats=new BuildStats();
```

13. Create a `Sensor1` object and a `Thread` object to run it.

```
Sensor1 sensor1=new Sensor1(stats);
Thread thread1=new Thread(sensor1,"Sensor 1");
```

14. Create a `Sensor2` object and a `Thread` object to run it.

```
Sensor2 sensor2=new Sensor2(stats);
Thread thread2=new Thread(sensor2,"Sensor 2");
```

15. Get the actual date and start the threads.

```
Date date1=new Date();

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

16. Wait for the finalization of the threads.

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

17. Write the results to the console.

```
    Date date2=new Date();
    stats.printStats();
    System.out.println("Execution Time: "+ ((date2.getTime()-
date1.getTime())/1000));
```

How it works...

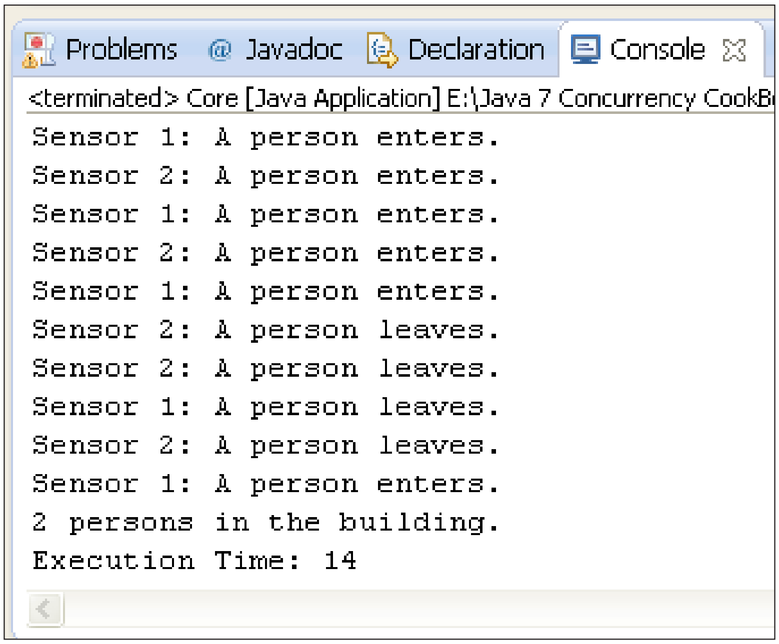
The example simulates an application that generates statistics about the people that come in or leave a building storing the number of people inside the building. This class has two methods: one to simulate when a person comes in and the other to simulate when a person goes out. Both methods call the auxiliary operations (the `generateCard()` and `generateReport()` methods) that have very long durations (two and three seconds respectively) relative to the rest of the method.

We have used the `synchronized` keyword to protect the access to the instruction that updates the number of persons in the building, leaving out of this block the long operations 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.

You have implemented two runnable classes, `Sensor1` and `Sensor2` that simulate two sensors that detect when people come in or go out, and updates the building statistics.

The main class of the example measures the running time of the runnable tasks.

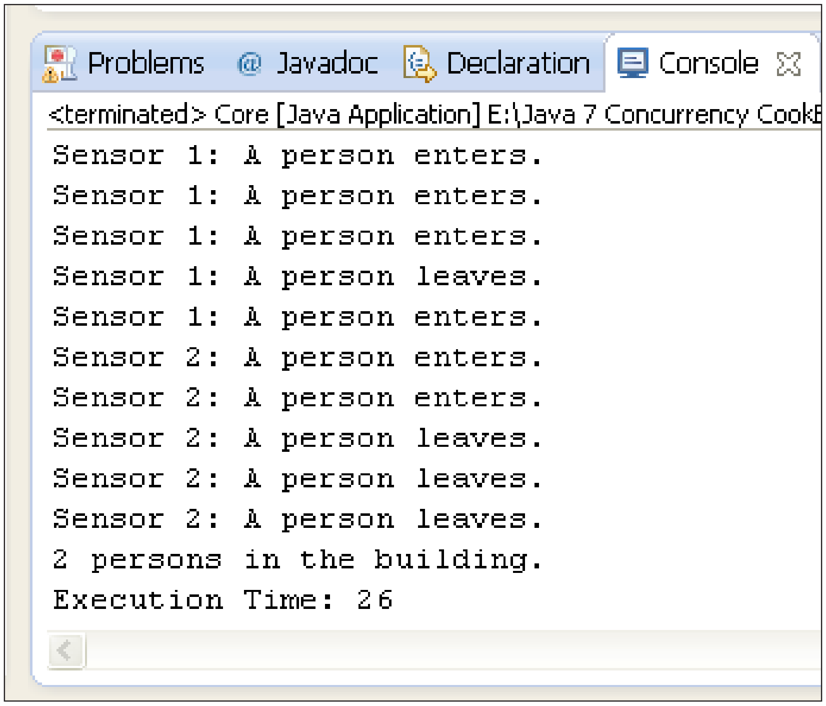
The following screenshot shows the output of an execution of the example:



Now, uncomment the `synchronized` keyword in the declaration of the methods and comment the keyword within the code. For example, this is the new `comeIn()` method:

```
public synchronized void comeIn(){
  System.out.printf("%s: A person enters.\n",Thread.currentThread().
getName());
  //synchronized(this) {
    numPeople++;
  //}
  generateCard();
}
```

In the following screenshot you can see the results of an execution of the modified application:



There is a big difference in the performance of the two versions of the application. You must be very careful with the use of the `synchronized` keyword so that it does not impact the performance of the application.

There's more...

As occurs with synchronized methods, a thread can call other synchronized methods and other methods with synchronized blocks of code protected by the same object. It doesn't need to get access to the synchronized blocks again.

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

See also

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

Processing results for Runnable objects in the Executor

The Executor framework allows the execution of concurrent tasks that returns a result using the `Callable` and `Future` interfaces. The traditional concurrent programming in Java is based on `Runnable` objects, but this kind of object doesn't return a result.

In this recipe, you will learn how to adapt a `Runnable` object to simulate a `Callable` one allowing a concurrent task to return a result.

Getting ready

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

How to do it...

Perform the following steps to implement the example:

1. Create a class named `FileSearch` and specify that it implements the `Runnable` interface. This class implements the file search operation.

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

2. Declare two private `String` attributes. One named `initPath` that will store the initial folder for the search operation. The other named `end` that will store the extension of the files this task is going to look for.

   ```
     private String initPath;
     private String end;
   ```

3. Declare a private `List<String>` attribute named `results` that will store the full path of the files that this task has found.

   ```
     private List<String> results;
   ```

4. Implement the constructor of the class that will initialize its attributes.

```
public FileSearch(String initPath, String end) {
  this.initPath = initPath;
  this.end = end;
  results=new ArrayList<>();
}
```

5. Implement the method `getResults()`. This method returns the list with the full paths of the files that this task has found.

```
public List<String> getResults() {
  return results;
}
```

6. Implement the `run()` method. First of all, write a log message to the console indicating that the task is starting its job.

```
@Override
public void run() {
  System.out.printf("%s: Starting\n",Thread.currentThread().
getName());
```

7. Then, if the `initPath` attribute stores the name of an existing folder, call the auxiliary method `processDirectory()` to process its files and folders.

```
  File file = new File(initPath);
  if (file.isDirectory()) {
    directoryProcess(file);
  }
```

8. Implement the auxiliary method `processDiretory()` that receives a `File` object as a parameter. First of all, get the content of the folder pointed by the parameter.

```
private void directoryProcess(File file) {
  File list[] = file.listFiles();
```

9. With all the elements of the folder, if they are folders, make a recursive call to the `processDirectory()` method. If they are files, call the `processFile()` auxiliary method.

```
  if (list != null) {
    for (int i = 0; i < list.length; i++) {
      if (list[i].isDirectory()) {
        directoryProcess(list[i]);
      } else {
        fileProcess(list[i]);
      }
    }
  }
```

10. Implement the auxiliary method `processFile()` that receives a `File` object with the full path of a file. This method checks if the file extension is equal to the one stored in the end attribute. If they are equal, add the full path of the file to the list of results.

```
private void fileProcess(File file) {
  if (file.getName().endsWith(end)) {
    results.add(file.getAbsolutePath());
  }
}
```

11. Implement a class named `Task` that extends the `FutureTask` class. You'll use `List<String> as the parameterized` type, as this will be the type of the data this task will return.

```
public class Task extends FutureTask<List<String>> {
```

12. Declare a private `FileSearch` attribute named `fileSearch`.

```
private FileSearch fileSearch;
```

13. Implement the constructor of this class. This constructor has two parameters: A `Runnable` object named `runnable` and a `List<String>` object named `result`. In the constructor, you have to call the constructor of the parent class passing to it the same parameters. Then, store the runnable parameter casting it to a `FileSearch` object.

```
public Task(Runnable runnable, List<String> result) {
  super(runnable, result);
  this.fileSearch=(FileSearch)runnable;
}
```

14. Override the `set()` method of the `FutureTask` class.

```
protected void set(List<String> v) {
```

15. If the parameter that it receives is null, store in it the result of calling the `getResults()` method of the `FileSearch` class.

```
  if (v==null) {
    v=fileSearch.getResults();
  }
```

16. Then, call the parent's method passing the received parameter as a parameter.

```
  super.set(v);
```

17. Finally, implement the main class of the example. Create a class named `Main` and implement the `main()` method.

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

18. Create a `ThreadPoolExecutor` object named `executor` calling the `newCachedThreadPool()` method of the `Executors` class.

    ```
        ThreadPoolExecutor executor=(ThreadPoolExecutor)Executors.
    newCachedThreadPool();
    ```

19. Create three `FileSearch` objects with a different initial folder for each one. You are going to look for files with the log extension.

    ```
        FileSearch system=new FileSearch("C:\\Windows", "log");
        FileSearch apps=new FileSearch("C:\\Program Files","log");
        FileSearch documents=new FileSearch("C:\\Documents And
    Settings","log");
    ```

20. Create three `Task` objects to execute the search operations in the executor.

    ```
        Task systemTask=new Task(system,null);
        Task appsTask=new Task(apps,null);
        Task documentsTask=new Task(documents,null);
    ```

21. Send these objects to the `executor` object using the `submit()` method. This version of the `submit()` method returns a `Future<?>` object, but you're going to ignore it. You have a class that extends the `FutureTask` class to control the execution of this task.

    ```
        executor.submit(systemTask);
        executor.submit(appsTask);
        executor.submit(documentsTask);
    ```

22. Call the `shutdown()` method of the `executor` object to indicate that it should finish its execution when these three tasks have finished.

    ```
        executor.shutdown();
    ```

23. Call the `awaitTermination()` method of the `executor` object indicating a long waiting period to guarantee that this method won't return until the three tasks have finished.

    ```
        try {
          executor.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
    ```

24. For each task, write a message with the size of the result list using the `get()` method of the `Task` object.

```
     try {
      System.out.printf("Main: System Task: Number of Results:
%d\n",systemTask.get().size());
      System.out.printf("Main: App Task: Number of Results:
%d\n",appsTask.get().size());
      System.out.printf("Main: Documents Task: Number of Results:
%d\n",documentsTask.get().size());
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
```

How it works...

The first point to take into consideration, to understand this example, is the difference between the `submit()` method of the `ThreadPoolExecutor` class when you pass a `Callable` object as the parameter and the `submit()` method when you pass a `Runnable` object as the parameter. In the first case, you can use the `Future` object that this method returns to control the status of the task and to get its result. But in the second case, when you pass a `Runnable` object, you can only use the `Future` object that this method returns to control the status of the task. If you call the `get()` method of that `Future` object, you will get the `null` value.

To override this behavior, you have implemented the `Task` class. This class extends the `FutureTask` class that is a class that implements the `Future` interface and the `Runnable` interface. When you call a method that returns a `Future` object (for example, the `submit()` method), you normally will get a `FutureTask` object. So you can use this class with two objectives.

- First, execute the `Runnable` object (in this case, a `FileSearch` object).
- Second, return the results that this task generates. To achieve this, you have overridden the `set()` method of the `Task` class. Internally, the `FutureTask` class controls when the task it has to execute has finished. At that moment, it makes a call to the `set()` method to establish the return value of the task. When you are executing a `Callable` object, this call is made with the value returned by the `call()` method, but when you are executing a `Runnable` object, this call is made with the `null` value. You have changed this `null` value with the list of results generated by the `FileSearch` object. The `set()` method will only have an effect the first time it is called. When it's called for the first time, it marks the task as finished and the rest of the calls will not modify the return value of the task.

In the `Main` class, instead of sending the `FutureTasks` objects to the `Callable` or `Runnable` objects, you can send it to the `executor` object. The main difference is that you use the `FutureTasks` objects to get the results of the task instead of the `Future` object returned by the `submit()` method.

In this case, you can still use the `Future` object returned by the `submit()` method to control the status of the task but remember that, as this task executes a `Runnable` object (you have initialized the `FutureTasks` objects with the `FileSearch` objects that implement the `Runnable` interface), if you call the `get()` method in the `Future` objects, you will get the `null` value.

There's more...

The `FutureTask` class provides a method not included in the `Future` interface. It's the `setException()` method. This method receives a `Throwable` object as the parameter and when the `get()` method is called, an `ExecutionException` exception will be thrown. This call has an effect only if the `set()` method of the `FutureTask` object hasn't been called yet.

See also

- The *Executing tasks in an executor that returns a result* recipe in *Chapter 4*, Thread Executors
- The *Creating and running a thread* recipe in *Chapter 1*, *Thread Management*

Processing uncontrolled exceptions in a ForkJoinPool class

The Fork/Join framework gives you the possibility to set a handler for the exceptions thrown by the worker threads of a `ForkJoinPool` class. When you work with a `ForkJoinPool` class, you should understand the difference between **tasks** and **worker threads**.

To work with the Fork/Join framework, you implement a task extending the `ForkJoinTask` class or, usually, the `RecursiveAction` or `RecursiveTask` classes. The task implements the actions you want to execute concurrently with the framework. They are executed in the `ForkJoinPool` class by the worker threads. A worker thread will execute various tasks. In the **work-stealing** algorithm implemented by the `ForkJoinPool` class, a worker thread looks for a new task when the task it was executing finishes its execution or it is waiting for the completion of another task.

In this recipe, you will learn how to process the exceptions thrown by a worker thread. You'll have to implement two additional elements for it to work as described in the following items:

- The first element is an extended class of the `ForkJoinWorkerThread` class. This class implements the worker thread of a `ForkJoinPool` class. You will implement a basic child class that will throw an exception.
- The second element is a factory to create worker threads of your own custom type. The `ForkJoinPool` class uses a factory to create its worker threads. You have to implement a class that implements the `ForkJoinWorkerThreadFactory` interface and uses an object of that class in the constructor of the `ForkJoinPool` class. The `ForkJoinPool` object created will use that factory to create worker threads.

How to do it...

Perform the following steps to implement the example:

1. First, implement your own worker thread class. Create a class named `AlwaysThrowsExceptionWorkerThread` that extends the `ForkJoinWorkerThread` class.

```
public class AlwaysThrowsExceptionWorkerThread extends
ForkJoinWorkerThread {
```

2. Implement the constructor of the class. It receives a `ForkJoinPool` class as a parameter and calls the constructor of its parent class.

```
  protected AlwaysThrowsExceptionWorkerThread(ForkJoinPool pool) {
    super(pool);
  }
```

3. Implement the `onStart()` method. This is a method of the `ForkJoinWorkerThread` class and is executed when the worker thread begins its execution. The implementation will throw a `RuntimeException` exception upon being called.

```
  protected void onStart() {
    super.onStart();
    throw new RuntimeException("Exception from worker thread");
  }
```

4. Now, implement the factory needed to create your worker threads. Create a class named `AlwaysThrowsExceptionWorkerThreadFactory` that implements the `ForkJoinWorkerThreadFactory` interface.

```
public class AlwaysThrowsExceptionWorkerThreadFactory implements
ForkJoinWorkerThreadFactory {
```

5. Implement the `newThread()` method. It receives a `ForkJoinPool` object as a parameter and returns a `ForkJoinWorkerThread` object. Create an `AlwaysThrowsExceptionWorkerThread` object and return it.

```
  @Override
  public ForkJoinWorkerThread newThread(ForkJoinPool pool) {
    return new AlwaysThrowsExceptionWorkerThread(pool);
  }
```

6. Implement a class that will manage the exceptions thrown by worker threads. Implement a class named `Handler` that implements the `UncaughtExceptionHandler` interface.

```
public class Handler implements UncaughtExceptionHandler {
```

7. Implement the `uncaughtException()` method. It receives as parameters a `Thread` object and a `Throwable` object and is called by the `ForkJoinPool` class each time a worker thread throws an exception. Write a message to the console and exit the program.

```
  @Override
  public void uncaughtException(Thread t, Throwable e) {
    System.out.printf("Handler: Thread %s has thrown an
Exception.\n",t.getName());
    System.out.printf("%s\n",e);
    System.exit(-1);
  }
```

8. Now, implement a task to be executed in the `ForkJoinPool` executor. Create a class named `OneSecondLongTask` that extends the `RecursiveAction` class.

```
public class OneSecondLongTask extends RecursiveAction{
```

9. Implement the `compute()` method. It simply sleeps the thread during one second.

```
  @Override
  protected void compute() {
    System.out.printf("Task: Starting.\n");
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("Task: Finish.\n");
  }
```

10. Now, implement the main class of the example. Create a class named `Main` with a `main()` method.

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

11. Create a new `OneSecondLongTask` object.

```
    OneSecondLongTask task=new OneSecondLongTask();
```

12. Create a new `Handler` object.

```
    Handler handler = new Handler();
```

13. Create a new `AlwaysThrowsExceptionWorkerThreadFactory` object.

```
    AlwaysThrowsExceptionWorkerThreadFactory factory=new
AlwaysThrowsExceptionWorkerThreadFactory();
```

14. Create a new `ForkJoinPool` object. Pass as parameters the value `2`, the `factory` object, the `handler` object, and the value `false`.

```
    ForkJoinPool pool=new ForkJoinPool(2,factory,handler,false);
```

15. Execute the task in the pool using the `execute()` method.

```
    pool.execute(task);
```

16. Shutdown the pool with the `shutdown()` method.

```
    pool.shutdown();
```

17. Wait for the finalization of the tasks using the `awaitTermination()` method.

```
    try {
      pool.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

18. Write a message indicating the end of the program.

```
    System.out.printf("Task: Finish.\n");
```

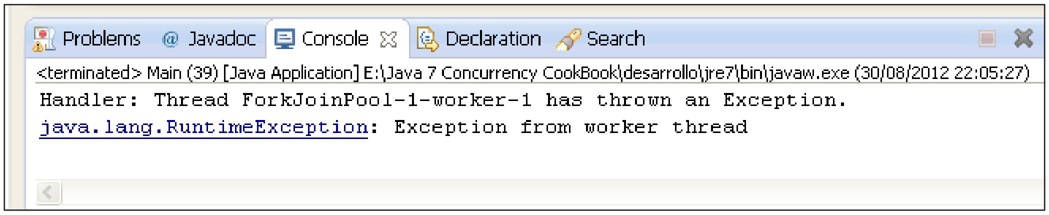
How it works...

In this recipe, you have implemented the following elements:

- **Your own worker thread class**: You have implemented the `AlwaysThrowsExceptionWorkerThread` class that extends the `ForkJoinWorkerThread` class that implements the worker threads of a Fork/Join pool. You have overridden the `onStart()` method. This method is executed when a worker thread starts its execution. It simply throws a `RuntimeException` upon being called.
- **Your own thread factory**: A `ForkJoinPool` class creates its worker threads using a factory. As you want to create a `ForkJoinPool` object that uses `AlwaysThrowsExceptionWorkerThreadFactory` worker threads, you have implemented a factory that creates them. To implement a worker thread factory, you have implemented the `ForkJoinWorkerThreadFactory` interface. This interface only has a method named `newThread()` that creates the worker thread and returns it to the `ForkJoinPool` class.
- **A task class**: The worker threads execute the tasks you send to the `ForkJoinPool` executor. As you want to start the execution of a worker thread, you have to send a task to the `ForkJoinPool` executor. The task sleeps for one second but, as the `AlwaysThrowsExceptionWorkerThread` thread throws an exception, it will never be executed.
- **A handler class for uncaught exceptions**: When a worker thread throws an exception, the `ForkJoinPool` class checks if an exception handler has been registered. You have implemented the `Handler` class for this purpose. This handler implements the `UncaughtExceptionHandler` interface that only has one method, that is, the `uncaughtException()` method. This method receives as a parameter the thread that throws the exception and the exception it throws.

In the `Main` class, you have put together all these elements. You have passed to the constructor of the `ForkJoinPool` class four parameters: the parallelism level, the number of active worker threads you want to have, the worker thread factory you want to use in the `ForkJoinPool` object, the handler you want to use for the uncaught exceptions of the worker threads, and the `async` mode.

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



When you execute the program, a worker thread throws a `RuntimeException` exception. The `ForkJoinPool` class hands it over to your handler, which in turn writes the message to the console and exits the program. The task doesn't start its execution.

There's more...

You can test two interesting variants of this example:

- If you comment the following line in the `Handler` class and execute the program, you will see a lot of messages written in the console. The `ForkJoinPool` class tries to start a worker thread to execute the task and, as it can't because they always throw an exception, it tries it over and over again.

  ```
  System.exit(-1);
  ```

- Something like that occurs if you change the third parameter (the exception handler) of the `ForkJoinPool` class constructor for the `null` value. In this case, you will see how the JVM writes the exceptions in the console.

  Take this into account when you implement your own worker threads that could throw exceptions.

See also

- The *Creating a Fork/Join pool* recipe in *Chapter 5*, *Fork/Join Framework*
- The *Customizing tasks running in the Fork/Join framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*
- The *Implementing the ThreadFactory interface to generate custom threads for the Fork/Join Framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*

Using blocking thread-safe lists for communicating with producers and consumers

The **producer/consumer problem** is a classical problem in concurrent programming. You have one or more producers of data that store this data into a buffer. You also have one or more consumers of data that takes the data from the same buffer. Both, producers and consumers, share the same buffer, so you have to control the access to it to avoid data inconsistency problems. When the buffer is empty, the consumers wait until the buffer has elements. If the buffer is full, the producers wait until the buffer has empty space.

This problem has been implemented using almost all techniques and synchronization mechanisms developed in Java and in other languages (refer to the *See Also* section to get more information). One advantage of this problem is that it can be extrapolated to a lot of real-world situations.

Java 7 concurrency API has introduced a data structure oriented to be used in these kinds of problems. It's the `LinkedTransferQueue` class and its main characteristics are as follows:

- It's a **blocking data structure**. The thread is blocked until the operation can be made, provided that the operations are performed immediately.
- Its size has no limit. You can insert as many elements as you want.
- It's a parameterized class. You have to indicate the class of the elements you're going to store in the list.

In this recipe, you will learn how to use the `LinkedTransferQueue` class running a lot of producer and consumer tasks that share a buffer of strings.

Getting ready...

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

How to do it...

Perform the following steps to implement the example:

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

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

2. Declare a private `LinkedTransferQueue` attribute parameterized with the `String` class named `buffer`.

    ```
      private LinkedTransferQueue<String> buffer;
    ```

3. Declare a private `String` attribute named `name` to store the name of the producer.

    ```
      private String name;
    ```

4. Implement the constructor of the class to initialize its attributes.

    ```
      public Producer(String name, LinkedTransferQueue<String> buffer)
    {
        this.name=name;
        this.buffer=buffer;
      }
    ```

5. Implement the `run()` method. Store 10,000 strings in the buffer using the `put()` method of the buffer object and write a message to the console indicating the end of the method.

    ```
      @Override
      public void run() {
        for (int i=0; i<10000; i++) {
          buffer.put(name+": Element "+i);
        }
        System.out.printf("Producer: %s: Producer done\n",name);
      }
    ```

6. Implement a class named `Consumer` and specify that it implements the `Runnable` interface.

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

7. Declare a private `LinkedTransferQueue` attribute parameterized with the `String` class named `buffer`.

    ```
      private LinkedTransferQueue<String> buffer;
    ```

8. Declare a private `String` attribute named `name` to store the name of the consumer.

```
private String name;
```

9. Implement the constructor of the class to initialize its attributes.

```
  public Consumer(String name, LinkedTransferQueue<String> buffer)
{
    this.name=name;
    this.buffer=buffer;
  }
```

10. Implement the `run()` method. Take out 10,000 strings from the buffer using the `take()` method of the buffer object and write a message to the console indicating the end of the method.

```
@Override
public void run() {
  for (int i=0; i<10000; i++){
    try {
      buffer.take();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  System.out.printf("Consumer: %s: Consumer done\n",name);
}
```

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

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

12. Declare a constant named `THREADS` and assign to it the value 100. Create a `LinkedTransferQueue` object with the `String` class object and call it `buffer`.

```
    final int THREADS=100;
    LinkedTransferQueue<String> buffer=new LinkedTransferQueue<
>();
```

13. Create an array of 100 `Thread` objects to execute 100 producer tasks.

```
    Thread producerThreads[]=new Thread[THREADS];
```

14. Create an array of 100 `Thread` objects to execute 100 consumer tasks.

```
    Thread consumerThreads[]=new Thread[THREADS];
```

15. Create and launch 100 `Consumer` objects and store the threads in the array created before.

```
for (int i=0; i<THREADS i++){
  Consumer consumer=new Consumer("Consumer "+i,buffer);
  consumerThreads[i]=new Thread(consumer);
  consumerThreads[i].start();
}
```

16. Create and launch 100 `Producer` objects and store the threads in the array created earlier.

```
for (int i=0; i<THREADS; i++) {
  Producer producer=new Producer("Producer: "+ I , buffer);
  producerThreads[i]=new Thread(producer);
  producerThreads[i].start();
}
```

17. Wait for the finalization of the threads using the `join()` method.

```
for (int i=0; i<THREADS; i++){
  try {
    producerThreads[i].join();
    consumerThreads[i].join();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

18. Write a message to the console with the size of the buffer.

```
    System.out.printf("Main: Size of the buffer: %d\n",buffer.
size());
    System.out.printf("Main: End of the example\n");
```

How it works...

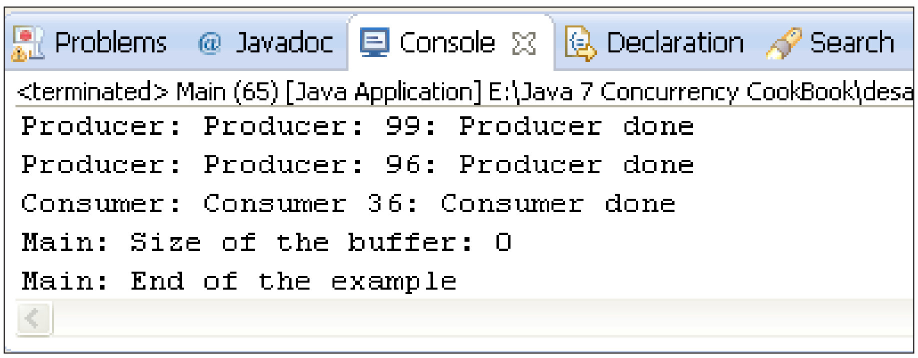
In this recipe, you have used the `LinkedTransferQueue` class parameterized with the `String` class to implement the producer/consumer problem. This `LinkedTransferQueue` class is used as a buffer to share the data between producers and consumers.

You have implemented a `Producer` class that adds strings to the buffer using the `put()` method. You have executed 100 producers and every producer inserts in the buffer 10,000 strings, so you insert 10,00,000 strings in the buffer. The `put()` method adds the element at the end of the buffer.

You also have implemented a `Consumer` class that gets a string from the buffer using the `take()` method. This method returns and deletes the first element of the buffer. If the buffer is empty, the method blocks the thread that makes the call until there are strings in the buffer to consume. You have executed 100 consumers and every consumer gets 10,000 strings from the buffer.

In the example, first, you have launched the consumers and then the producers, so, as the buffer is empty, all the consumers will be blocked until the producers begin their execution and stores strings in the list.

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



To write the number of elements of the buffer, you have used the `size()` method. You have to take into account that this method can return a value that is not real, if you use them 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 `LinkedTransferQueue` class provides other useful methods. These are some of them:

- `getWaitingConsumerCount()`: This method returns the number of consumers that are blocked in the `take()` method or `poll (long timeout, TimeUnit unit)` because the `LinkedTransferQueue` object is empty.
- `hasWaitingConsumer()`: This method returns `true` if the `LinkedTransferQueue` object has consumers waiting or `false` otherwise.

- `offer(E e)`: This method adds the element passed as a parameter at the end of the `LinkedTransferQueue` object and returns the `true` value. `E` represents the class used to parameterize the declaration of the `LinkedTransferQueue` class or a subclass of it.
- `peek()`: This method returns the first element in the `LinkedTransferQueue` object, but it doesn't delete it from the list. If the queue is empty, the method returns the `null` value.
- `poll(long timeout, TimeUnit unit)`: This version of the poll method, if the `LinkedTransferQueue` buffer is empty, waits for the specified period of time for it. If the specified period of time passes and the buffer is still empty, 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 *Using conditions in synchronized code* recipe in *Chapter 2*, *Basic Thread Synchronization*
- The *Changing data between concurrent tasks* recipe in *Chapter 3*, *Thread Synchronization Utilities*

Monitoring a Thread class

Threads are the most basic element of the Java Concurrency API. Every Java program has at least one thread that executes the `main()` method that, in turn, starts the execution of the application. When you launch a new `Thread` class, it's executed in parallel to the other threads of the application and with the other processes on an operating system. There is a critical difference between process and thread. A process is an instance of an application that is running, (for example, you're editing a document in a text processor). This process has one or more threads that execute the tasks that makes the process. You can be running more than one process of the same application, for example, two instances of the text processor.

All the kinds of Java tasks that you can execute (`Runnable`, `Callable`, or `Fork/Join` tasks) are executed in threads and all the advanced Java concurrency mechanisms, as the `Executor` **framework** or the `Fork/Join` **framework**, are based on pools of threads.

In this recipe, you will learn what information you can obtain about the status of a `Thread` 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 another IDE like NetBeans, open it and create a new Java project.

How to do it...

Perform the following steps to implement the example:

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

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

2. Implement the `run()` method of the task.

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

3. Create a loop with 100 steps.

```
    for (int i=0; i<100; i++) {
```

4. In each step, put the thread to sleep for 100 milliseconds.

```
      try {
        TimeUnit.MILLISECONDS.sleep(100);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
```

5. Write a message in the console with the name of the thread and the step number.

```
      System.out.printf("%s: %d\n",Thread.currentThread().
getName(),i);
    }
  }
}
```

6. Create the main class of the example. Create a class named `Main` with a `main()` method.

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

7. Create a `Task` object named `task`.

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

8. Create a `Thread` array with five elements.

```
    Thread thread[] = new Thread[5];
```

9. Create and start five threads to execute the `Task` object created before.

```
    for (int i = 0; i < 5; i++) {
      thread[i] = new Thread(task);
      thread[i].setPriority(i + 1);
      thread[i].start();
    }
```

10. Create a loop with ten steps to write information about the threads launched before. Inside it, create other loop with five steps.

```
for (int j = 0; j < 10; j++) {
  System.out.printf("Main: Logging threads\n");
  for (int i = 0; i < threads.length; i++) {
```

11. For each thread, write in the console its name, its status, its group, and the length of its stack trace.

```
        System.out.printf("**********************\n");
        System.out.printf("Main: %d: Id: %d Name: %s: Priority:
%d\n",i,threads[i].getId(),threads[i].getName(),threads[i].
getPriority());
        System.out.printf("Main: Status: %s\n",threads[i].
getState());
        System.out.printf("Main: Thread Group: %s\n",threads[i].
getThreadGroup());
        System.out.printf("Main: Stack Trace: \n");
```

12. Write a loop to write the stack trace of the thread.

```
        for (int t=0; t<threads[i].getStackTrace().length; t++) {
          System.out.printf("Main: %s\n",threads[i].
getStackTrace()[t]);
        }
        System.out.printf("**********************\n");
      }
Sleep the thread for one second and close the loop and the class.
      TimeUnit.SECONDS.sleep(1);
    }
  }
}
```

How it works...

In this recipe, you have used the following methods to get information about a `Thread` class:

- `getId()`: This method returns the ID of a thread. It's a unique long number and it can't be changed.
- `getName()`: This method returns the name of a thread. If you don't establish the name of the thread, Java gives it a default name.
- `getPriority()`: This method returns the priority of execution of a thread. Threads with higher priority are executed in preference to threads with lower priority. It's an `int` value that has a value between the constants `MIN_PRIORITY` and `MAX_PRIORITY` of the `Thread` class. By default, threads are created with the same priority, specified by the constant `NORM_PRIORITY` of the `Thread` class.

- `getState()`: This method returns the status of a thread. It's a `Thread.State` object. The `Thread.State` enumeration has all the possible states of a thread.
- `getThreadGroup()`: This method returns the `ThreadGroup` object of a thread. By default, threads belong to the same thread group, but you can establish a different one in the constructor of a thread.
- `getStackTrace()`: This method returns an array of `StackTraceElement` objects. Each of these objects represent a call to a method that begins with the `run()` method of a thread and includes all the methods that have been called until the actual execution point. When a new method is called, a new stack trace element is added to the array. When a method finishes its execution, its stack trace element is removed from the array.

There's more...

The `Thread` class includes other methods that provide information about it that can be useful. These methods are as follows:

- `activeCount()`: This method returns the number of active threads in a group of threads.
- `dumpStack()`: This method prints the stack trace of a thread to the standard error output.

See also

- The *Creating and running a thread* recipe in *Chapter 1*, *Thread Management*
- The *Using a ThreadFactory interface in an Executor framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*
- The *Implementing a ThreadFactory interface to generate custom threads for the Fork/Join framework* recipe in *Chapter 7*, *Customizing Concurrency Classes*

Monitoring a Semaphore class

A **semaphore** is a counter that protects the access to one or more shared resources.

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

When a thread wants to use shared resources, it must acquire a semaphore. If the internal counter of the semaphore is greater than 0, the semaphore decrements the counter and allows the access to the shared resource. If the counter of the semaphore is 0, the semaphore blocks the thread until the counter is greater than 0. When the thread has finished using the shared resource, it must release the semaphore. That operation increases the internal counter of the semaphore.

In Java, semaphores are implemented in the `Semaphore` class.

In this recipe, you will learn what information you can obtain about the status of a semaphore and how to obtain it.

Getting ready

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

How to do it...

Perform the following 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 `Semaphore` attribute named `semaphore`.

   ```
     private Semaphore semaphore;
   ```

3. Implement the constructor of the class to initialize its attribute.

   ```
     public Task(Semaphore semaphore){
       this.semaphore=semaphore;
     }
   ```

4. Implement the `run()` method. First, acquire a permit of the `semaphore` attribute writing a message in the console to indicate that circumstance.

   ```
     @Override
     public void run() {
       try {
         semaphore.acquire();
         System.out.printf("%s: Get the semaphore.\n",Thread.
   currentThread().getName());
   ```

5. Then, put the thread to sleep for two seconds using the `sleep()` method. Finally, release the permit and write a message in the console to indicate that circumstance.

```
      TimeUnit.SECONDS.sleep(2);
      System.out.println(Thread.currentThread().getName()+":
Release the semaphore.");
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      semaphore.release();
    }
```

6. Implement the main class of the example. Create a class named `Main` with a `main()` method.

```
public class Main {

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

7. Create a `Semaphore` object named `semaphore` with three permits.

```
    Semaphore semaphore=new Semaphore(3);
```

8. Create an array to store ten `Thread` objects.

```
    Thread threads[]=new Thread[10];
```

9. Create and start ten `Thread` objects to execute ten `Task` objects. After starting a thread, put the thread to sleep for 200 milliseconds and call the `showLog()` method to write information about the `Semaphore` class.

```
    for (int i=0; i<threads.length; i++) {
      Task task=new Task(semaphore);
      threads[i]=new Thread(task);
      threads[i].start();

      TimeUnit.MILLISECONDS.sleep(200);

      showLog(semaphore);
    }
```

10. Implement a loop with five steps to call the `showLog()` method to write information about the semaphore and put the thread to sleep for one second.

```
    for (int i=0; i<5; i++) {
      showLog(semaphore);
      TimeUnit.SECONDS.sleep(1);
    }
  }
```

11. Implement the `showLog()` method. It receives a `Semaphore` object as a parameter. Write in the console information about the available permits, queued threads, and permits of the semaphore.

```
  private static void showLog(Semaphore semaphore) {
    System.out.printf("********************\n");
    System.out.printf("Main: Semaphore Log\n");
    System.out.printf("Main: Semaphore: Avalaible Permits: 
%d\n",semaphore.availablePermits());
    System.out.printf("Main: Semaphore: Queued Threads: 
%s\n",semaphore.hasQueuedThreads());
    System.out.printf("Main: Semaphore: Queue Length: 
%d\n",semaphore.getQueueLength());
    System.out.printf("Main: Semaphore: Fairness: %s\n",semaphore.
isFair());
    System.out.printf("********************\n");
  }
```

How it works...

In this recipe, you have used the following methods to get information about a semaphore:

- `availablePermits()`: This method returns an `int` value which is the number of available resources of a semaphore.
- `hasQueuedThreads()`: This method returns a `Boolean` value indicating if there are threads waiting for a resource protected by a semaphore.
- `getQueueLength()`: This method returns the number of threads that are waiting for a resource protected by a semaphore.
- `isFair()`: This method returns a `boolean` value indicating if a semaphore has the fair mode activated. When the fair mode is active (this method returns the `true` value), and the lock has to select another thread to give to it the access to the shared resource, it selects the longest-waiting thread. If the fair mode is inactive (this method returns the `false` value), there is no guarantee about the order in which threads are selected to get the access to the shared resource.

See also

- The *Controlling concurrent access to a resource* recipe in *Chapter 3*, *Thread Synchronization Utilities*
- The *Controlling concurrent access to multiple copies of a resource* recipe in *Chapter 3*, *Thread Synchronization Utilities*