# Exceptions

# 14

**Outcomes**:

*By the end of this chapter you should be able to*:

- *explain the term* ***exception***;
- *distinguish between* ***checked*** *and* ***unchecked*** *exception classes in Java*;
- *claim an exception using a* ***throws*** *clause*;
- *throw an exception using a* ***throw*** *command*;
- *catch an exception in a* ***try…catch*** *block*;
- *use a* ***finally*** *block or a* ***try-with-resources*** *construct to deal with clean-up issues*;
- *use the* `Optional` *class to avoid* `NullPointerException` *errors*;
- *define and use their own exception classes.*

## 14.1 Introduction

One way in which to write a program is to assume that everything proceeds smoothly and as expected—users input values at the correct time and of the correct format, files are never corrupt, array indices are always valid and so on. Of course this view of the world is very rarely accurate. In reality, unexpected situations arise that could compromise the correct functioning of your program.

Programs should be written that continue to function even if unexpected situations should arise. So far we have tried to achieve this by carefully constructed **`if`** statements that send back error flags, in the form of **`boolean`** values, when appropriate. However, in some circumstances, these forms of protection against undesirable situations prove inadequate. In such cases the *exception handling* facility of Java must be used.



Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_14

## 14.2 Pre-defined Exception Classes in Java

An **exception** is an event that occurs during the life of a program which could cause that program to behave unreliably. You can see that the events we described in the introduction fall into this category. For example, accessing an array with an invalid index could cause that program to terminate.

Each type of event that could lead to an exception is associated with a pre-defined *exception class* in Java. When a given event occurs, the Java run-time environment determines which exception has occurred and an object of the given exception class is generated. This process is known as **throwing** an exception.

These exception classes have been named to reflect the nature of the exception. For example, when an array is accessed with an illegal index an object of the `ArrayIndexOutOfBoundsException` class is thrown.

All exception classes inherit from the base class `Throwable` which is found in the `java.lang` package. These subclasses of `Throwable` are found in various packages and are then further categorized depending upon the type of exception. For example, the exception associated with a given file not being found (`FileNotFoundException`) and the exception associated with an end of file having been reached (`EOFException`) are both types of input/output exceptions (`IOException`), which reside in the `java.io` package. Figure 14.1 illustrates part of this hierarchy.

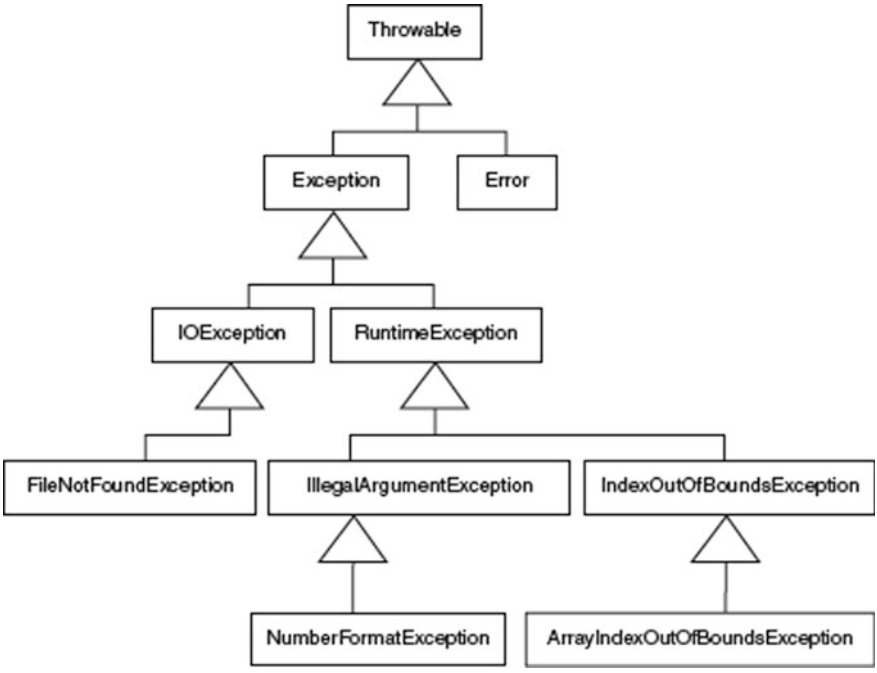



**Fig. 14.1** A sample of Java's pre-defined exception class hierarchy

As you can see from Fig. 14.1, there are two immediate subclasses of `Throwable`: `Exception` and `Error`. The `Error` class describes internal system errors that are very unlikely ever to occur (so called "hard" errors). For example, one subclass of `Error` is `VirtualMachineError` where some error in the JVM has been detected. There is little that can be done in the way of recovery from such errors other than to end the program as gracefully as possible. All other exceptions are subclasses of the `Exception` class and it is these exceptions that programmers deal with in their programs. The `Exception` class is further subdivided. The two most important subdivisions are shown in Fig. 14.1, `IOException` and `RuntimeException`.

The `RuntimeException` class deals with errors that arise from the logic of a program. For example, a program that converts a string into a number, when the string contains non-numeric characters (`NumberFormatException`) or accesses an array using an illegal index (`ArrayIndexOutOfBoundsException`).

The `IOException` class deals with external errors that could affect the program during input and output. Such errors could include, for example, the keyboard locking, or an external file being corrupted.

Since nearly every Java instruction could result in a `RuntimeException` error, the Java compiler does not flag such instructions as potentially error-prone. Consequently these types of errors are known as unchecked exceptions. It is left to the programmer to ensure that code is written in such a way as to avoid such exceptions; for example, checking an array index before looking up a value in an array with that index. Should such an exception arise, it will be due to a program error and will not become apparent until runtime.

The Java compiler *does*, however, flag up those instructions that may generate all other types of exception (such as `IOException` errors) since the programmer has no means of avoiding such errors arising. For example, an instruction to read from a file may cause an exception because the file is corrupt. No amount of program code can prevent this file from being corrupt. The compiler will not only flag such an instruction as potentially error-prone, it will also specify the exact exception that could be thrown. Consequently, these kinds of errors are known as checked exceptions. Programmers have to include code to inform the compiler of how they will deal with checked exceptions generated by a particular instruction, before the compiler will allow the program to compile successfully.

## 14.3 Handling Exceptions

Consider a simple program that allows the user to enter an aptitude test mark at the keyboard; the program then informs the user if he or she has passed the test and been allowed on a given course. We could use the `nextInt` method (from either our `EasyScanner` class, or the original `Scanner` class) to allow the user to enter this mark. However, in order to show you how exceptions can be dealt with in your programs, we will not take this approach—instead we will devise our own class,

`TestException`, that will contain a class method called `getInteger`. Before we do that, here is the outline of the main application:

```
public class TestException
{
    // this method is declared 'static' as it is a class method
    public static int getInteger()
    {
            // code for method goes here
    }
}
```

Now let's look at an outline for the `TestException` class.

The `getInteger` method must allow the user to enter an integer at the keyboard and then return that integer. There are many ways we could try and read an integer from the keyboard. As we have said, rather than make use of the `nextInt` method in the `Scanner` class, the approach we will take here will be to use a rather low-level method called `read` in the `System.in` object. So far we have used the `System.out` object to display information on the screen, but we have not explored the `System.in` object. This object is an object of the `InputStream` class that you will find out more about in Chap. 18.

You will remember from Chap. 2 that each character on the keyboard is represented by a Unicode number. For countries in which the standard western alphabet is used, the lower case letters 'a' through to 'z' are represented by the Unicode values 97 through to 122 inclusive. Special characters also have Unicode values. For example, the carriage return character has a Unicode value 13. The `InputStream` class provides a `read` method that is a bit like the `next` method of the `Scanner` class, except that it treats the string as a series of Unicode numbers. Each number is considered to be of type byte, so that the string itself is an array of bytes. Figure 14.2 illustrates the effect of the `read` method when someone enters the word "hello" at the keyboard.

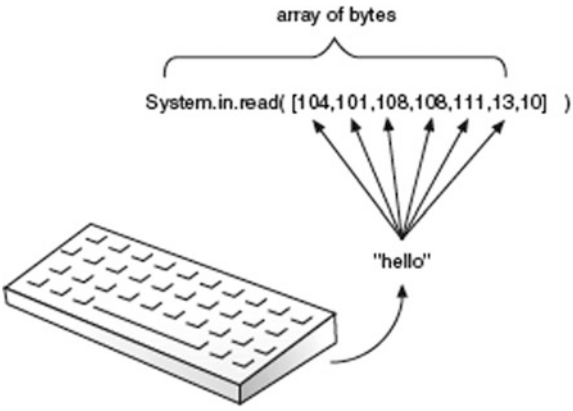



**Fig. 14.2** The 'read' method stores characters entered at the keyboard as an array of bytes

Notice that the array of bytes is not returned as a value but instead sent as a parameter. Also note that the new-line character is given a Unicode value of 10.

The `getInteger` method will first have to take this array of bytes and convert it into a `String`. Luckily a version of the `String` constructor returns a `String` object from an array of bytes. We then remove any trailing spaces at the end of the `String`; this can be done with the `String` method `trim` as follows:

```
byte [] buffer = new byte[512]; // declare a large byte array
System.in.read(buffer); // characters entered stored in array
String s = new String (buffer); // make string from byte array
s = s.trim(); // trim string
```

Now, finally, we have to convert this string into an integer. We can use the `parseInt` method of the `Integer` class to do this:

```
int num = Integer.parseInt(s); // converts string to an 'int'
```

Our `TestException` class now looks like this:

```
// this is a first attempt, it will not compile!
public class TestException
{
  public static int getInteger()
   {
            byte [] buffer = new byte[512];
            System.in.read(buffer);
            String s = new String (buffer);
            s = s.trim();
            int num = Integer.parseInt(s);
            return num; // send back the integer value
   }
}
```

Unfortunately, as things stand, this class will *not* compile. The cause of the error is in the `getInteger` method, in particular the way we used the `read` method of `System.in`. Whenever this method is used, the Java compiler *insists* that we be very careful. To understand this better, take a look at the header for this `read` method, taken from the Java language specification, in particular the part we have emboldened:

```
public int read (byte[] b) throws IOException
```

Up until now you have not seen a method header of this form. The words **`throws`** `IOException` are the new bits in this method header. In Java this is known as a method **claiming an exception**.

### 14.3.1 Claiming an Exception

The term claiming an exception refers to a given method having been marked to indicate that it will pass on an exception object that it might generate. So the term **`throws`** `IOException` means that the method *could* cause an input/output exception in your program. The type of error that could take place while data is being read includes the loss of a network connection or a file being corrupted, for example.

Remember, when an exception occurs, an exception object is created. This is an unwanted object that could cause your program to fail or behave unpredictably, and so should be dealt with and not ignored. Rather than dealing with this exception object *within* the `read` method, the Java people decided it would be better if callers of this method dealt with the exception object in whatever way they felt was suitable. In effect, they *passed the error* on to the caller of the method (see Fig. 14.3).

As the type of exception generated (`IOException`) is not a subclass of `RuntimeException`, it is an example of a *checked exception*. In other words, the compiler *insists* that if the `read` method is used, the programmer deals with this exception in some way, and does not just ignore it as we did originally. That is why we had a compiler error initially. There are always two ways to deal with an exception:

1. deal with the exception within the method;
2. pass on the exception out of the method.

The developers of the `read` method decided to pass on the exception, so now our `getInteger` method has to decide what to do with this exception. In a while we will show you how to deal with an exception within a method, but for now we will just make our `getInteger` method pass on the exception too. We do this by simply adding a **`throws`** clause to our method:

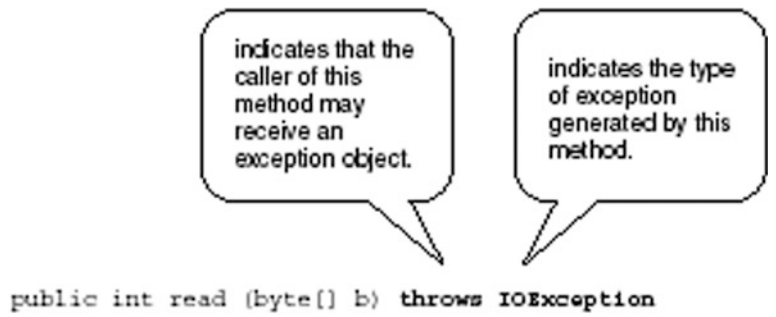



**Fig. 14.3** The 'throws' clause can be added to a method header to indicate that the method may generate an exception

```
import java.io.IOException;
public class TestException
{
   // adding this throws clause will allow this method to compile
   public static int getInteger() throws IOException
   {
       // as before
   }
}
```

Notice that, as the `IOException` class is in the `io` package, we now need the following **`import`** statement at the top of this class:

```
import java.io.IOException;
```

Now that the `getInteger` method has claimed the `IOException`, it *will* compile as we have not just *ignored* the exception, we have made a *conscious decision* to pass the exception on to any method that calls this `getInteger` method. Now, let's think about developing our aptitude test program.

```
// something wrong here!
public class AptitudeTest
{
   public static void main (String[] args)
   {
       int score;
       System.out.print("Enter aptitude test score: ");
       score = TestException.getInteger(); // calling class method
       // test score here
   }
}
```

Can you see what the problem with this program is? Well, this program will not compile now as the `main` method makes a call to our `getInteger` method, and this method may now throw an `IOException`! The `main` method now has to deal with this exception and not just ignore it. For the time being, to keep the compiler happy, we will just let the `main` method throw this exception as well. Here is the code:

***AptitudeTest***

```
import java.io.IOException;
public class AptitudeTest
{
      // this main method will throw out any IOExceptions
      public static void main (String[] args) throws IOException
      {
            int score;
            System.out.print("Enter aptitude test score: ");
            // the 'getInteger' method may throw an IOException
            score = TestException.getInteger();
            if (score >= 50)
            {
                  System.out.println("You have a place on the course!");
            }
            else
            {
                  System.out.println("Sorry, you failed your test");
            }
      }
}
```

Dealing with the exception in the way we have is not a very good idea. We have effectively continually passed on the exception object until it gets thrown out of our program to the operating system. This may cause the program to terminate when such an exception occurs. Before we deal with this problem let us show you a test run. Take a look at it as something very interesting happens.

```
Enter aptitude test score: 12w
java.lang.NumberFormatException: 12w
at java.lang.Integer.parseInt(Integer.java:418)
at java.lang.Integer.parseInt(Integer.java:458)
at TestException.getInteger(TestException.java:10)
at AptitudeTest.main(AptitudeTest.java:11)
```

As you can see, when asked to enter an integer, the user inadvertently added a character into the number (12**w**). This has led to an exception being generated and thrown out of our program. Again, looking at the output generated, you can see that when an exception is generated in this way the Java system gives you quite a lot of information. This information includes the name of the method that threw the exception, the class that the method belongs to, the line numbers in the source files where the error arose, and the type of exception that was thrown. Such information is referred to as the **stack trace** of the exception.

Look at the name of the exception that is thrown. It's not the one we were worried about, `IOException`, but `NumberFormatException`. This exception is raised when trying to convert a string into a number when the string contains non-numeric characters, as we were trying to do in this case within our `getInteger` method:

```
public static int getInteger() throws IOException
{
    // some code here
    int num = Integer.parseInt(s); /* will cause a NumberFormatException
                                   if string s contains non-numeric characters*/
}
```

Why didn't the compiler warn us about this when we first used the `parseInt` method in our implementation of `getInteger`? Well, the reason is that the exception that could arise (`NumberFormatException`) is a subclass of `RunTimeException` and so is unchecked!

Notice that run-time exceptions do not need to be claimed in method headers in order for them to be thrown. For example, although the following is valid in Java, it is not *necessary* to claim the `NumberFormatException` in the header.

```
/* multiple exceptions can be claimed in the method header as follows by separating exception
  names with commas. However run-time exceptions do not need to be claimed in this way */

public static int getInteger() throws IOException, NumberFormatException
{
      // some code here
}
```

The way we have dealt with exceptions so far has not been very effective. As you can see from the test run of program 14.1, continually throwing exceptions up to the calling method does not really solve the problem. It may keep the compiler happy, but eventually it means exception objects will escape from your programs and cause them to terminate. Instead, it is better at some point to handle an exception object rather than throw it. In Java this is known as **catching an exception**.

### 14.3.2 Catching an Exception

One route for an exception object is *out* of the current method and *up to* the calling method. That's the approach we used in the previous section. Another way out for an exception object, however, is into a **`catch`** block. Once an exception object is trapped in a **`catch`** block, and that block ends, the exception object is effectively terminated. In order to trap the exception object in a **`catch`** block you must surround the code that could generate the exception in a **`try`** block. The syntax for using a **`try`** and **`catch`** block is as follows:

```
try
{
    // code that could generate an exception
}
catch (Exception e) // type of exception must be specified as a parameter
{
    // action to be taken when an exception occurs
}
// other instructions could be placed here
```

There are a few things to note before we show you this **`try`**…**`catch`** idea in action. First, any number of lines could be placed within the **`try`** block, and more than one of them could cause an exception. If none of them causes an exception the **`catch`** block is missed and the lines following the **`catch`** block are executed. If any one of them causes an exception the program will leave the **`try`** block and look for a **`catch`** block that deals with that exception.

Once such a **`catch`** clause is found, the statements within it will be executed and the program will then *continue* with any statements that follow the **`catch`** clause—it will *not* return to the code in the **`try`** clause. Look carefully at the syntax for the **`catch`** block:

```
catch (Exception e)
{
     // action to be taken when an exception occurs
}
```

**Table 14.1** Some methods of the *Exception* class

| Method | Description |
|---|---|
| `printStackTrace` | Prints (onto the console) a stack trace of the exception |
| `toString` | Returns a detailed error message |
| `getMessage` | Returns a summary error message |

This looks very similar to a method header. You can see that the **`catch`** block header has one parameter: an object, which we called `e`, of type `Exception`. Since *all* exceptions are subclasses of the `Exception` class, this will catch *any* exception that should arise. However, it is better to replace this exception class with the *specific* class that you are catching so that you can be certain *which* exception you have caught. As there may be more than one exception generated within a method, there may be more than one **`catch`** block below a **`try`** block—each dealing with a different exception. When an exception is thrown in a **`try`** block, the **`catch`** blocks are inspected in order—the first matching **`catch`** block is the one that will handle the exception.

Within the **`catch`** block, programmers can, if they choose, interrogate the exception object using some `Exception` methods, some of which are listed in Table 14.1.

With this information in mind we can deal with the exceptions in the previous section in a different way. All we have to decide is where to catch the exception object. For now we will leave the `getInteger` method as it is, and catch offending exception objects in the `main` method of the `AptitudeTest2` program. Take a look at it and then we will discuss it.

***AptitudeTest2***

```
import java.io.IOException;

public class AptitudeTest2
{
    public static void main (String[] args)
    {
       try
       {
              int score;
              System.out.print("Enter aptitude test score: ");
              // getInteger may throw IOException or NumberFomatException
              score = TestException.getInteger();
              if (score >= 50)
              {
                     System.out.println("You have a place on the course!");
              }
              else
              {
                     System.out.println("Sorry, you failed your test");
              }
       }
       // if something does goes wrong!
       catch (NumberFormatException e)
       {
              System.out.println("You entered an invalid number!");
       }
       catch (IOException e)
       {
              System.out.println(e); // calls toString method
       }
       // even if no exception thrown/caught, this line will be executed
       System.out.println("Goodbye");
   }
}
```

Notice that by catching an offending exception object there is no need to pass that object out of the method by raising that exception in the method header. Since we catch the `IOException` here, the `throws IOException` clause can be removed from the header of `main`. In this program we have chosen to print out an error message if an `IOException` is raised (by implicitly calling the `toString` method), whereas we have chosen to print our own message if a `NumberFormatException` is raised. Now look at a sample test run of the program:

```
Enter aptitude test score: 12w
You entered an invalid number!
Goodbye
```

As you can see the user once again entered an invalid integer, but this time the program did not terminate. Instead the exception was handled with a clear message to the user, after which the program continued to operate normally.

## 14.4 The 'finally' Clause

From the previous sections you can see that three courses of action may now take place in a **`try`** block:

1. the instructions within the **`try`** block are all executed successfully;
2. an exception occurs within the **`try`** block; the **`try`** block is exited and a matching **`catch`** block is found for this exception;
3. an exception occurs within the **`try`** block; the **`try`** block is exited but no matching **`catch`** block is found for this exception, so the exception is thrown from the method.

It may be the case that, no matter which of these courses of action take place, you wish to execute some additional instructions before the method terminates. Often such a scenario arises when you wish to carry out some clean-up code, such as closing a file or a network connection that you have opened in the **`try`** block. We will see some examples of these scenarios in later chapters. The **`finally`** clause allows us to do this. The syntax for the **`finally`** clause is as follows:

```
try
{
       // code that could generate an exception
}
catch (Exception e) /* if one or more 'catch' clauses are specified,
                       they must be given before the 'finally' clause */
{
       // action to be taken when an exception occurs
}
finally
{
       // cleanup code can go here
}

// other instructions could be placed here
```

Notice that when **`catch`** clauses are specified, the **`finally`** clause must come directly *after* such clauses. If no such **`catch`** clauses are specified, the **`finally`** clause must follow directly after the **`try`** clause. Now, when the code in the **`try`** block is executed the following three courses of action can take place:

1. the instructions within the **`try`** block are all executed successfully; if there are any **`catch`** blocks specified they are skipped and the code in the **`finally`** block is executed, followed by any code after the **`finally`** block;
2. an exception occurs within the **`try`** block; the **`try`** block is exited and a matching **`catch`** block is found for this exception, after which the code in the **`finally`** block is executed, followed by any code after the **`finally`** block;
3. an exception occurs within the **`try`** block; the **`try`** block is exited but no matching **`catch`** block is found for this exception so the code in the **`finally`** block is executed—after which the method terminates and the appropriate exception is thrown by the given method.

We will use a very simple example of closing a `Scanner` resource as a way to demonstrate how the **`finally`** clause works under the three scenarios outlined above.

We have seen many examples of creating `Scanner` objects. `Scanner` objects point to a resource, in our case this resource has been the keyboard (`System.in`). We would not ordinarily close this resource as this would be closed automatically once the program terminates, but in the `ClosingAResourceUsingFinally` program below we will close it explicitly in a **`finally`** clause. Take a look at the code and then we will discuss it.

***ClosingAResourceUsingFinally***

```
import java.util.Scanner;

public class ClosingAResourceUsingFinally
{
    public static void main(String[] args)
    {
       Scanner keyboard = new Scanner (System.in); // open a Scanner resource
       try
       {
            System.out.println("START TRY");
            String[] colours = {"RED","BLUE","GREEN"}; // initialise array
            System.out.print("Which colour? (1,2,3): ");
            String pos = keyboard.next();
            // next line could throw NumberFormatException
            int i = Integer.parseInt(pos);
            // next line could throw ArrayIndexOutOfBoundsException
            System.out.println(colours[i-1]);
            System.out.println("END TRY");
        }
       // include a catch only for ArrayIndexOutOfBoundsException
       catch(ArrayIndexOutOfBoundsException e)
       {
            System.out.println("ENTER CATCH ");
            System.out.println(e);
       }
       // this block will always be executed
       finally
       {
            System.out.println("ENTER FINALLY");
            keyboard.close(); // Scanner resource closed
            System.out.println("Scanner CLOSED");
       }
       System.out.println("Goodbye");

    }
}
```

Most of this code should be self-explanatory. Notice that we have provided a **catch** block for the `ArrayIndexOutOfBoundsException` but we have not provided a **catch** block for the `NumberFormatException`, so such an exception would be thrown from `main` should it arise. Also, notice that we displayed messages to indicate when we are in each of the **try**, **catch** and **finally** blocks. Finally, notice how we close a `Scanner` resource, in the **finally** block of code, by calling the `Scanner` object's `close` method:

```
// this block will always be executed
finally
{
       System.out.println("ENTER FINALLY");
       keyboard.close(); // Scanner resource closed
       System.out.println("Scanner CLOSED");
}
```

Here is one test run:

```
START TRY
Which colour? (1,2,3): 2
BLUE
END TRY
ENTER FINALLY
Scanner CLOSED
Goodbye
```

Here the user enters a valid colour number, so the **try** block completes successfully. The **catch** block is skipped and the **finally** block is executed, which closes the `Scanner` resource. Following the **finally** block the last "Goodbye" instruction is executed.

Here is another test run:

```
START TRY
Which colour? (1,2,3): 4
ENTER CATCH
java.lang.ArrayIndexOutOfBoundsException: 3
ENTER FINALLY
Scanner CLOSED
Goodbye
```

Here the user enters an invalid colour number, the **try** block does not complete as an `ArrayIndexOutOfBoundsException` is thrown. A matching **catch** block is found for this exception and executed. Upon completion of this **catch** block the program continues with the code in the **finally** block, which closes the `Scanner` resource. Following the **finally** block the last "Goodbye" instruction is executed.

Here is the last test run:

```
START TRY
Which colour? (1,2,3): 2c
ENTER FINALLY
Scanner CLOSED
Exception in thread "main" java.lang.
NumberFormatException:
For input string: "2c"
at java.lang.NumberFormatException.forInputString
(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:456)
at java.lang.Integer.parseInt(Integer.java:497)
at
ClosingAResourceUsingFinally.main
(ClosingAResourceUsingFinally.java:11)
```

In this case, the user entered an invalid number causing a `NumberFormatException`—so the **`try`** block did not complete successfully. However, there is no **`catch`** block provided for this exception. Without a **`finally`** clause this would have led to program termination *immediately* as the exception escapes from `main`. We have a **`finally`** clause, however, which closes the `Scanner` resource. The program then terminates with the offending exception (`NumberFormatException`).

These three test runs match the three scenarios we identified for the **`try`**/**`catch`**/**`finally`** blocks earlier. You will notice from the three test runs above that, if the instructions inside the **`finally`** clause were written as normal below the **`catch`** clause (without putting them into a **`finally`** block), the first two test runs would have produced exactly the same result. This is because code following a **`catch`** block is always executed if no exception is thrown, or if an exception is thrown and a matching **`catch`** clause is found and executed. Only in scenario three, when an exception is thrown and no matching **`catch`** is found (perhaps because no **`catch`** clauses were specified), does the **`finally`** clause really make a difference to program flow.

You may well come across the third scenario when developing your programs, and we shall see examples in later chapters, so the **`finally`** clause could be used here for clean-up code. Using the **`finally`** clause in scenarios one and two is optional.

## 14.5 The 'Try-with-Resources' Construct

In the previous section we saw how a **`finally`** clause can be used to close a resource before exiting a program. In the test runs of `ClosingAResourceUsingFinally` we saw that, once the **`finally`** code is executed, any uncaught exception is reported

(`NumberFormatException` in the test runs of Sect. 14.4). However, you should note that if the code in the **`finally`** clause itself throws an exception it is *this* exception that is thrown from the method rather than the original offending exception. The `close` method of `Scanner`, for example, would itself throw an `IOException` if the system was unable to close the `Scanner` resource. This is not ideal as the original exception is probably more appropriate to report.

One of the more recent developments in Java allows for us to avoid writing **`finally`** clauses to close a resource, but instead adapt the **`try`** clause for this purpose. This is known as **try-with-resources**. The *try-with-resources* clause automatically closes a specified resource (or resources) for us and suppresses any exceptions that might arise from doing so, leaving any original uncaught exceptions free to be reported. The program below re-writes the `ClosingAResourceUsingFinally` program to make use of this try-with-resources clause. Take a look at it and then we will discuss it.

***ClosingAResourceUsingTryWithResources***

```
import java.util.Scanner;

public class ClosingAResourceUsingTryWithResources
{
   public static void main(String[] args)
   {
      try (Scanner keyboard = new Scanner (System.in)) // open a Scanner resource here
      {
            System.out.println("START TRY");
            String[] colours = {"RED","BLUE","GREEN"}; // initialise array
            System.out.print("Which colour? (1,2,3): ");
            String pos = keyboard.next();
            // next line could throw NumberFormatException
            int i = Integer.parseInt(pos);
            // next line could throw ArrayIndexOutOfBoundsException
            System.out.println(colours[i-1]);
            System.out.println("END TRY");
        }
      // include a catch only for ArrayIndexOutOfBoundsException
      catch(ArrayIndexOutOfBoundsException e)
      {
            System.out.println("ENTER CATCH ");
            System.out.println(e);
      }
      // we have removed the finally clause
      System.out.println("Goodbye");

   }
}
```

You can see that, when using a try-with-resources clause, we include the resource declaration (in this case the creation of a `Scanner` object) in round brackets straight after the try keyword:

```
try (Scanner keyboard = new Scanner (System.in)) // Scanner resource created in brackets
{
      // previous code as before
}
```

This Scanner resource is now treated as an object whose scope is the **try** block.

If code in the **try** block completes, with no exception occurring, the given resource (our Scanner object keyboard in this case) is automatically closed. If an exception does occur during the **try** block, the given resource is still automatically closed, whether or not that exception is caught. Unlike a **finally** block, however, if an exception is thrown from the **try** block but not caught and an exception is then also thrown by the close method of the given resource, the exception thrown by the close method is suppressed and the original uncaught exception from the **try** block is reported.

So, in the example above, if a NumberFormatException occurs that has no **catch** block and closing the Scanner resource also throws an IOException, the IOException is suppressed and the NumberFormatException is thrown.

Note, we can create any object (or objects) within the **try-with-resources** bracket as long its type is a class that implements the Closeable (or AutoCloseable) interfaces. Classes that implement these interfaces all include a close method. Scanner is one example of such a class. We will see further examples Chap. 18 when we study file-handling.

## 14.6 Null-Pointer Exceptions

One of the most common exception types you will come across when programming is the dreaded NullPointerException. This is an example of an unchecked RuntimeException and is a very common cause of program errors. It occurs when we try to call the method of an object, but the object itself contains a **null** value rather than valid object data. As a very simple example, consider the following instructions that make use of our BankAccount class of Chap. 8:

```
BankAccount acc1 = null; // BankAccount reference set to null
System.out.println(acc1.getAccountName()); // 'getAccountName' will cause a NullPointerException
```

Here the BankAccount object reference, acc1, is set to the **null** value. We should not call any methods on this object as it is not yet referencing a valid BankAccount object. However, we have then made an attempt to call the getAccountNumber method on this object reference and, consequently, the program will throw a NullPointerException.

While we are unlikely to make such obvious errors as the one demonstrated above, **null** values can find their way into your programs in very subtle ways and NullPointerExceptions can then easily arise.

Think back to the `Bank` collection class of Sect. 8.8.1, for storing a collection of `BankAccount` objects. Here is a reminder of its `getItem` method that retrieves a `BankAccount` object given its account number.

```
// returns the account assciated with a particular account number
public BankAccount getItem(String accountNumberIn)
{
   int index = search(accountNumberIn);
   if(index != -999) // check that account exists
   {
        return list.get(index); // return valid BankAccount object
   }
   else
   {
        return null; // no such account so return null value
   }
}
```

You can see that, if the given account number exists in the list, the associated `BankAccount` object is returned:

```
if(index != -999) // check that account exists
{
     return list.get(index); // return valid BankAccount object
}
```

If, however, if no such account exists a **`null`** value is returned to indicate this:

```
else
{
     return null; // no such account so return null value
}
```

Now consider the following simple program that adds two `BankAccount` objects to the `Bank` collection class and then asks the user to enter an account number to search for:

***NullPointerDemo***

```
public class NullPointerDemo
{
      public static void main(String[] args)
      {
            // create Bank collection
            Bank myBank = new Bank();
            // add two BankAccount objects
            myBank.addAccount("001", "Aaron");
            myBank.addAccount("002", "Quentin");
            // request user for account number
            System.out.print("Enter account number to search for: ");
            String account = EasyScanner.nextString(); // we are using our EasyScanner class here
            // retrieve account name accociated with this account number
            // this may throw a NullPointerException!
            System.out.println(myBank.getItem(account).getAccountName());

      }
}
```

Here is a sample program where the user enters a valid account number to search for:

```
Enter account number to search for: 001
Aaron
```

Once a valid account number is entered, "001" in this case, the appropriate account name ("Aaron") is displayed.

Here is a sample program run with an invalid account number entered:

```
Enter account number to search for: 003
Exception in thread "main" java.lang.NullPointerException
     at NullPointerDemo.main(NullPointerDemo.java:15)
```

As you can see, in this case, having entered an invalid account number ("003") a `NullPointerException` is thrown. Here is the offending instruction:

```
System.out.println(myBank.getItem(account).getAccountName());
```

The problem here is that, as we have seen, the `getItem` method (highlighted in bold) does not return a valid `BankAccount` object but, instead, a **`null`** value when the given account number is not present in the list. Consequently, the attempt to get the account number via the `getAccountNumber` fails with a `NullPointerException` then thrown.

Of course, we can avoid this error by using an **`if`** statement to check for a **`null`** value and only call the `getAccoutNumber` method if we do not have a **`null`** value:

```
if (myBank.getItem(account) != null) // add a check for non-null value
{
       System.out.println(myBank.getItem(account).getAccountName()); // safe to call method
}
```

While this would work, unfortunately very often these **`if`** checks are not included and potential `NullPointerExceptions` can remain. To deal with this problem, Java has recently introduced the `Optional` class to better deal with such `NullPointerExceptions`.

## 14.7 The *Optional* Class

The `Optional` class is essentially a wrapper class that can contain the value of a given type, or it can contain **`null`**. The best way to see how this `Optional` class works is to use it in an example, so let's re-write the `getItem` method in our `Bank` collection class to make use of this new class.

Firstly, the `Optional` class resides in the `util` folder, so we will need the following **`import`** statement at the top of our `Bank` class:

```
import java.util.Optional;

public class Bank
{
       // code for Bank class goes here
}
```

Now, here is the new `getItem` method; take a look at it and then we will discuss it:

```
//note the Optional return type
public Optional<BankAccount> getItem(String accountNumberIn)
{
        int index = search(accountNumberIn);
        if(index != -999) // check that account exists
        {
            return Optional.of(list.get(index)); // send valid Optional value
        }
        else
        {
            return Optional.empty(); // send empty(null) Optional value
        }
}
```

Firstly, you can see that the return type is no longer just `BankAccount`, but an `Optional` value that *could* contain a `BankAccount`—the generics mechanism is used to fix the contained type:

```
// Optional return type may contain a BankkAccount or null
public Optional<BankAccount> getItem(String accountNumberIn)
{
       // code for getItem here
}
```

We should point out here that changing the `getItem` method in this way would mean that we would also have to change the `deposit` and `withdraw` methods, both of which call `getItem`; we do this by using the `get` method of `Optional`, as explained below.

The `Optional` return type makes it clear to anyone using this class that the value being returned could contain a `BankAccount` *or* it could contain **`null`**, whereas a return type of `BankAccount` would not have made this clear.

Now, when we wish to return a `BankAccount` object we wrap it up inside an `Optional` value using the `Optional` class method `of` as follows:

```
if(index != -999) // check that account exists
{
       return Optional.of(list.get(index)); // send valid Optional value
}
```

To indicate no BankAccount object can be found we wrap up a **null** value inside an Optional value using the Optional class method empty as follows:

```
else
{
        return Optional.empty(); // send empty(null) Optional value
}
```

Programmers are no longer able to treat the returned value as a BankAccount object. It is now an object of type Optional<BankAccount>. So, for example, the following instruction from our NullPointerDemo program will no longer compile:

```
System.out.println(myBank.getItem(account).getAccountName()); //compiler error
```

Since getItem now returns an Optional<BankAccount> object we are no longer able to call a BankAccount method such as getAccountNumber, and attempting to do so raises a compiler error.

So how do we avoid this compiler error? One approach would be to check if a valid value has been returned. We can use the isPresent method of the Optional class, that returns **true** if a valid object is returned and **false** otherwise, on the object returned:

```
if (myBank.getItem(account).isPresent())
{
        // code to retrieve valid object here
}
```

Now can retrieve the BankAccount object before we call the getAccount-Name method. We can use the get method of the Optional class to do this:

```
if (myBank.getItem(account).isPresent())
{
        // notice use of 'get' method to retrieve the BankAccount object inside Optional
        System.out.println(myBank.getItem(account).get().getAccountName());
}
```

There is also another method we can use called ifPresent. This receives an object of the functional interface Consumer, which you came across in Chap. 13. It is one of the "out-of-the-box" interfaces and was described in Table 13.1. Its abstract method, accept, receives a single parameter (of whatever type is chosen for the generic interface) and its return type is void. We can therefore call ifPresent with a lambda expression. Below, we have re-written the NullPointerDemo program using this syntax; take a look at it and then we will discuss it:

***NullPointerDemoWithOptional***

```
public class NullPointerDemoWithOptional
{
    public static void main(String[] args)
    {
       // create Bank collection
       Bank myBank = new Bank();  // version with new getItem method
       // add two BankAccount objects
       myBank.addAccount("001", "Aaron");
       myBank.addAccount("002", "Quentin");

       // request user for account number
       System.out.print("Enter account number to search for: ");
       String account = EasyScanner.nextString(); // we are using our EasyScanner class here

       // retrieve account name accociated with this account number using lambda, and display result
       myBank.getItem(account).ifPresent(value -> System.out.println(value.getAccountName()));
    }
}
```

You can see that we have retrieved a `BankAccount` using the new version of `getItem`, and it will therefore be of type `Optional<BankAccount>`. We then call the `ifPresent` method of `Optional` with the correct lambda expression for the abstract method of the `Consumer` interface. The input to this method (which we have called `value`) is the item held in the `Optional` object, which will be either a `BankAccount` or a **`null`** value. The `ifPresent` method executes the lambda expression only if the item held is a valid object—if it is not, it does nothing.

```
myBank.getItem(account).ifPresent(value -> System.out.println(value.getAccountName()));
```

Thus, the program displays the account name if a valid object was retrieved, otherwise nothing is displayed.

You can see how the `Optional` type alerts programmers that the **`null`** value might be returned and ensures they unpack this value before proceeding, thus avoiding `NullPointerExceptions`. It also provides a variety of methods that, potentially, avoid the need for **`null`** value **`if`** checks in your code. There are a variety of other methods contained in the `Optional` class, which you can look up on the Oracle™ site.

## 14.8 Exceptions in GUI Applications

In Sect. 14.4 we showed you how the `parseInt` method could potentially result in a `NumberFormatException` being thrown. If this were not handled at some point, the exception object would escape out of your program and cause the program to terminate.

However, this isn't the first time that you used the `parseInt` method. You often had to use it when implementing your JavaFX GUI applications. In such applications all user input would normally be retrieved as strings, and then converted to numbers by calling methods such as `parseInt` method and `parseDouble`.

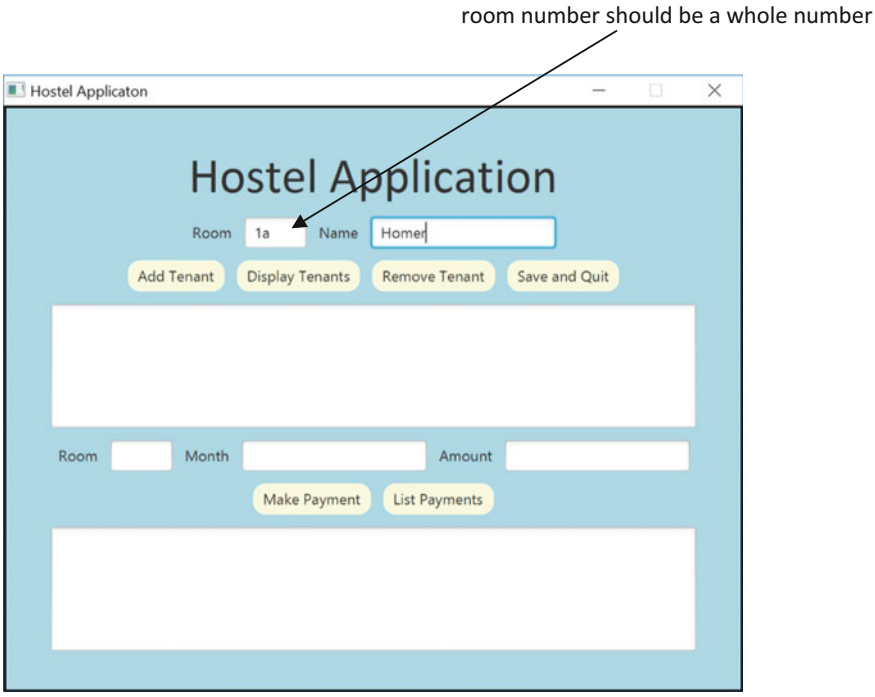



**Fig. 14.4** A sample screen shot from the 'Hostel' case study illustrating an invalid room number having been entered

At the time, you never considered handling these exceptions, and your applications never seemed to terminate as a result of invalid data entry. For example, do you remember the `Hostel` case study of Chaps. 11 and 12? Figure 14.4 illustrates a sample screen shot when a user enters an invalid room number.

When such an event occurred within your JavaFX application, the application seemed to continue operating regardless. After our discussion on exceptions this might seem surprising as the text entered is being processed by a `parseInt` method. To remind you, here is a fragment from the `addHandler` method:

```
private void addHandler()
{
        String roomEntered =  roomField1.getText(); // code to read room number
        String nameEntered =  nameField.getText();

       // previous additional code here

       // code to check room number, parseInt could cause an exception!
       if(Integer.parseInt(roomEntered)< 1 || Integer.parseInt(roomEntered)>noOfRooms)
       {
            displayArea1.setText ("There are only "  + noOfRooms  + " rooms");
       }

       // code to addTenant here
}
```

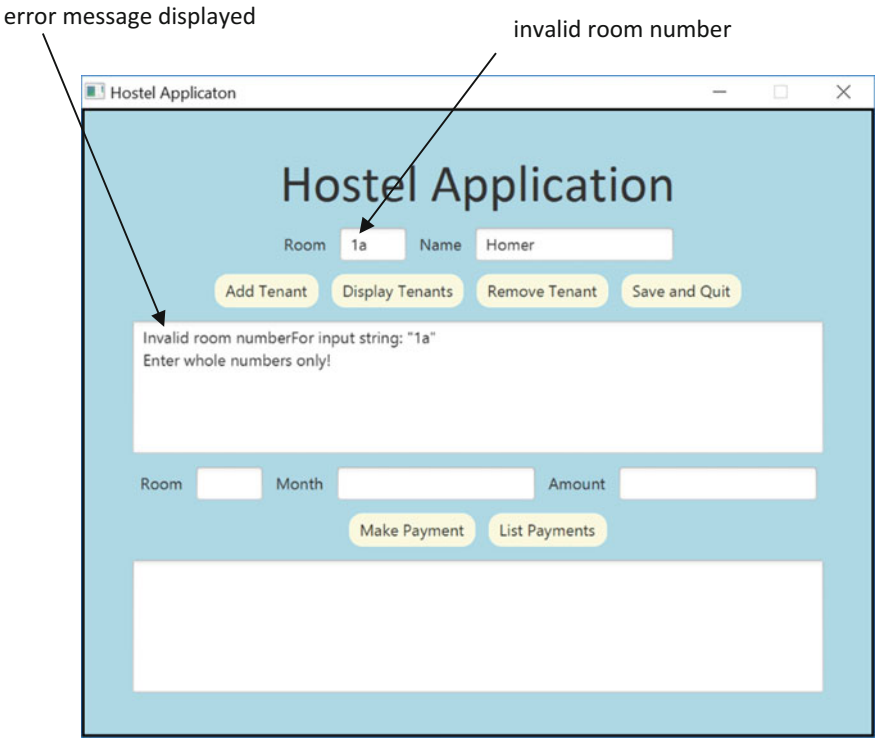



**Fig. 14.5** Exceptions can still be dealt with in JavaFX applications

In fact, when an invalid number is entered as illustrated in Fig. 14.4, a `NumberFormatException` occurs—but:

- you will not see details of the exception in your graphics screen since they will always be displayed in the output window of your IDE, or, if you are running your program from the command line, on the black console window (which may be hidden during the running of your application);
- exceptions do not terminate JavaFX applications; however, they may make them behave unpredictably.

So, if you look at the console screen or output window, you will see a list of exceptions that have been thrown during the running of your JavaFX applications—you may be surprised to see how many are actually thrown when you thought your program was operating correctly.

Often, graphical programs will continue to operate normally in the face of exceptions. To ensure this is the case you should still add exception handling code into your JavaFX applications. For example, we could amend the event-handler above as follows:

```
private void addHandler()
{
       try // place previous code in try block
       {
              // previous add handler code here
       }
       catch (NumberFormatException e) // catch exception and diplay error message on GUI
       {
              displayAreaI.setText("Invalid room number" + e.getMessage()
                                                  + "\nEnter whole numbers only!");

       }
}
```

Now if we run the application again, with the same input as depicted in Fig. 14.4, we get the response given in Fig. 14.5.

## 14.9 Using Exceptions in Your Own Classes

So far we have mainly been dealing with how to handle predefined exceptions (as summarised in Fig. 14.1) that are automatically thrown by your Java programs, such a `NullPointerException` being thrown when calling a method of a **`null`** value.

Sometimes, however we wish to generate an error under circumstances when Java would not ordinarily raise an error. For example, think back to the `PaymentList` collection class from the case study in Chap. 11. Here is an outline of that class:

```
import java.util.ArrayList;

public class PaymentList
{
        private ArrayList<Payment> pList;
        public final int MAX;

        /** Constructor initialises the empty payment list and sets the maximum list size
         * @param   maxIn The maximum number of payments in the list
         */

       public PaymentList(int maxIn)
       {
         pList = new ArrayList<>();
         MAX = maxIn;
       }

       // remaining methods go here
```

Here we have an `ArrayList` to hold the payments and a `MAX` value to record the maximum number of payments. There are no Java exceptions we need to be concerned about here. But take a look at the code for the constructor. Can you see a potential problem here?

We are setting the value of `MAX` and this value is sent as an integer. We would want this number to be a positive number, but integers in Java can be positive or negative or zero. If a negative or zero value is sent, Java will still allow us to set

MAX to this value but doing so would create knock on logical problems in our program code.

Ideally, we should avoid setting MAX to a value that is less than 1, and would instead wish to report an error. We could try and use an **if else** statement to check the parameter, maxIn, and then try and report an error if need be:

```
public PaymentList(int maxIn)
{
	pList = new ArrayList<>();
	if (maxIn < 1) // check if parameter is not positive
	{
		// report error
	}
	else
	{
		MAX = maxIn; // ok to set MAX
	}
}
```

How would we report back this error? One way, which we have used before, would be to return a **boolean** value of **false** to indicate failure. A general problem with reporting errors using **boolean** values is that these values can be ignored. In this case we have another problem—constructors can have no return value! The only way to report back errors from constructors is to throw exceptions.

### 14.9.1 Throwing Exceptions

We have seen circumstances where our programs automatically throw exceptions, but we have not seen examples of where we throw our own exceptions. Let's take a look at the syntax to do this.

The first thing we need to decide is *which* exception to throw. We could make use of one of the pre-existing Java exception class. Since the error we wish to report is a logical error we could try throwing a RuntimeException. In order to throw an exception object you must:

- write an instruction explicitly to throw the exception using a **throw** command;
- combine this with the **new** command to generate an object of the appropriate exception type by calling the given exception class's constructor.

Below is a modified PaymentList constructor that throws a RuntimeException:

```
public PaymentList(int maxIn)
{
	pList = new ArrayList<>();
	if (maxIn < 1)
	{
		throw new RuntimeException( ); // force a RuntimeException to be thrown
	}
	else
	{
		MAX = maxIn;
	}
}
```

Notice this version of the `RuntimeException` constructor requires no parameters:

```
throw new RuntimeException( ); //  no parameters required here
```

Every Java exception class is provided with an alternative version of the constructor that can receive a `String` parameter, often used to bundle information regarding the error within the exception object.

Let's use this alternative version of the `RuntimeException` constructor to provide some useful error information:

```
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
              throw new RuntimeException("invalid list size set " + maxIn ); // send error info
       }
       else
       {
              MAX = maxIn;
       }
}
```

Now let's look at a program that makes use of this modified `PaymenList` constructor, catches the `RuntimeException` if it is thrown and displays this exception object (containing our error message).

***RuntimeExceptionDemo***

```
public class RuntimeExceptionDemo
{
    public static void main(String[] args)
    {
        try
        {
            System.out.print("Enter size of list: ");
            int size = EasyScanner.nextInt();  // we are using our EasyScanner here
            PaymentList p = new PaymentList(size); // can now throw RuntimeException
        }
        catch(RuntimeException e) // catch exception from PaymentList constructor
        {
            System.out.println(e); // display exception object
        }
        System.out.println("END OF DEMO");

    }
}
```

First, a normal test run:

```
Enter size of list: 5
END OF DEMO
```

Now a test run with an invalid list size:

```
Enter size of list: 0
java.lang.RuntimeException: invalid list size set 0
END OF DEMO
```

Here you can see an illegal list size of zero triggers the `RuntimeException` and displaying this exception object reveals our error information.

In the example above we had an error that made sense to our program, avoiding using a non-positive value when setting the maximum size of our list, but we have co-opted an existing exception name to report this error (`RuntimeException`).

Using a pre-existing exception type name (such as `RuntimeException`) to report your own program-specific errors has a few problems. Firstly, a clause to catch this exception will also catch any other exception of the given type. However, you might want very different error handling code for your program specific errors. Secondly, the name of the exception type is not a good description of your specific error. It would be better if we could invent a new name (or names) for your program-specific errors. For example, if you were developing a game where pieces moved on a board you might want to have an exception called `InvalidMoveException` rather than use a generic name such as `RuntimeException`.

Luckily, it is very simple to create your own exception classes with names of your choice.

### 14.9.2 Creating Your Own Exception Classes

You can create your own exception class by inheriting from any predefined exception class. Generally speaking, if you wish your exception class to be unchecked then it should inherit from `RuntimeException` (or one of its sub-classes), whereas if you wish your exception to be checked you can inherit from the general `Exception` class.

In the case of the `PaymentList` class we wish to make the exception thrown by the constructor unchecked, as it is a logical error not an input/output error. So we will define our exception class by inheriting from the `RuntimeException` class. We will call this new exception class `HostelException` so it can be used throughout our Hostel application if need be. Look at the code first and then we will discuss it.

***HostelException***

```
public class HostelException extends RuntimeException
{
       public HostelException () // constructor without parameter
       {
              super("error in Hostel application");
       }

       public HostelException (String message)// constructor with parameter
       {
              super (message);
       }
}
```

As well as inheriting from some exception class, user-defined exception classes should have two constructors defined within them. One should take no parameters and simply call the constructor of the superclass with a default message of your choosing:

```
public HostelException () // constructor without parameter
{
        super("error in Hostel application"); // default error info
}
```

The other constructor should allow a user-defined message to be supplied with the exception object:

```
public HostelException (String message)// constructor with parameter
{
        super (message); // user defined message can be provided
}
```

The `PaymentList` constructor can now be modified to make use of this `HostelException` class as follows:

```
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
              throw new HostelException("invalid list size set " + maxIn ); // user-defined error
       }
       else
       {
              MAX = maxIn;
       }
}
```

Finally, we can amend the tester so that the new `HostelException` is caught:

***HostelExceptionDemo***

```
public class HostelExceptionDemo
{
    public static void main(String[] args)
    {
        try
        {
            System.out.print("Enter size of list: ");
            int size = EasyScanner.nextInt();
            PaymentList p = new PaymentList(size); // can now throw HostelException
        }
        catch(HostelException e) // catch exception from PaymentList constructor
        {
            System.out.println(e); // display exception object
        }
        catch (Exception e) // note general catch clause added
        {
            System.out.println("Some unforseen error");
            e.printStackTrace();
        }
        System.out.println("END OF DEMO");
    }
}
```

Notice how we added a general **`catch`** clause to catch any exceptions that we might not yet have considered. In this **`catch`** clause we have printed the stack trace in this error-handler to determine the exact cause of this unexpected error:

```
catch (Exception e) // catches all uncaught errors
{
      System.out.println("Some unforseen error");
      e.printStackTrace();
}
```

During testing this is always a good strategy. However, you need to ensure that the general **`catch`** clause is the *last* **`catch`** clause you specify. For example, something like the following will not compile:

```
try
{
      // some code here
}
catch (Exception e) // catches all uncaught errors
{
      // some code here
}
catch (HostelException e) // will not compile!
{
      // some code here
}
```

The above will not compile as the first general **`catch`** clause (**`catch`** `Exception`) will catch *all* exception types (including `HostelException`). So, any **`catch`** clauses below (such as catch `HostelException`) will never be reached. To write unreachable code in Java causes a compiler error.

## 14.10 Documenting Exceptions

We finish off this chapter by looking at how to document methods that may throw exceptions, using the `Javadoc` style of comments discussed in Chap. 11. The `@throws` tag should be used to document the name of any exceptions that may be thrown by a method. Here for example, is the `PaymentList` constructor, documented with `Javadoc` comments:

```
/** Constructor initialises the empty payment list and sets the maximum list size
 * @param   maxIn              The maximum number of payments in the list
 * @throws  HostelException    If the list is sized with a non-positive value
 */
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
             throw new HostelException("invalid list size set " + maxIn);
       }
       MAX = maxIn;
}
```

Generally speaking, when documenting methods in this way, it is good practice to document *all* exceptions that a method may throw. Multiple `@throws` tags can be used to list multiple exceptions.

## 14.11 Self-test Questions

1. What is an *exception*?
2. Distinguish between *checked* and *unchecked* exceptions and then identify which of the following exceptions are checked, and which are unchecked:

   - `FileNotFoundException`;
   - `NegativeArraySizeException`;
   - `NullPointerException`;
   - `NumberFormatException`;
   - `IOException`;
   - `Exception`;
   - `ArrayIndexOutOfBoundsException`;
   - `RuntimeException`.

3. Explain the following terms:

   (a) *claiming* an exception;
   (b) *throwing* an exception;
   (c) *catching* an exception.

4. Look at the program below and then answer the questions that follow:

```
public class ExceptionsQ4
{
   public static void main(String[] args)
   {
       int[] someArray = {12,9,3,11};
       int position = getPosition();
       display (someArray, position);
       System.out.println("End of program" );
    }

    static int getPosition()
    {
       System.out.println("Enter array position to display");
       String positionEntered = EasyScanner.nextString(); // requires EasyScanner class
       return Integer.parseInt(positionEntered);
    }

    static void display (int[] arrayIn, int posIn)
    {
        System.out.println("Item at this position is: " + arrayIn[posIn]);
    }
}
```

(a) Will this result in any compiler errors?

(b) Which methods could throw exceptions?

(c) Identify the names of the exceptions that could be thrown and the circumstances under which they could be thrown.

5. What is the purpose of a **`finally`** clause?

6. When would you use the *try-with-resources* construct?

7. Consider once again the `PaymentList` class of Chap. 11. In particular here is a reminder of its `getItem` method that returns a particular payment given a position in the list:

```
/** Reads the payment at the given position in the list
 *  @param      positionIn The logical position of the payment in the list
 *  @return     Returns the payment at the given logical position in the list
 *              or null if no payment at that logical position
 */
public Payment getPayment(int positionIn)
{
      //check for valid logical position
      if (positionIn <1 || positionIn > getTotal())
      {
            // no object found at given position
            return null;
      }
      else
      {
            // take one off logical position to get ArrayList position
            return pList.get(positionIn - 1);
      }
}
```

(a) Explain the purpose of the `Optional` class.

(b) Re-write the `getPayment` method using an `Optional` class.

8. When would it be appropriate to define your own exception class?

## 14.12 Programming Exercises

1. Implement the program given in self-test question 4 above. Now:

(a) Re-write `main` so that it catches any exceptions it may now throw by displaying a message on the screen indicating the exception thrown.

(b) At the moment, the "End of program" message may not always be executed. Add an appropriate **`finally`** clause so that this message is always executed at the end of the program.

(c) Add an additional **`catch`** clause in `main` to catch any unaccounted-for exceptions (within this **`catch`** clause print out the stack trace of the exception).

(d) Create your own exception class `InvalidPositionException` (make this an unchecked exception).

(e) Re-write the display method so that it throws the `InvalidPositionException`.

(f) Re-write `main` to take account of this amended display method.

(g) Document these exceptions using appropriate `Javadoc` comments.

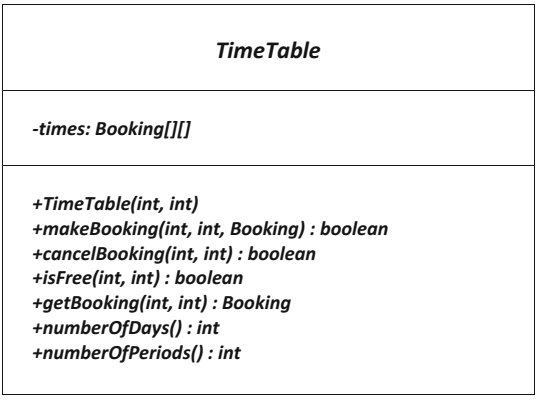
2. The `Scanner` class has methods `nextInt` and `nextDouble` for reading an **`int`** and a **`double`** value from the keyboard respectively. Both of these methods throw an exception if an appropriate numerical value is not entered.

(a) Write a tester program to find out the name of this exception.

(b) Develop a new version of the `EasyScanner` class, say `EasyScannerPlus`, so that instead of throwing exceptions the methods `nextInt` and `nextDouble` repeatedly display an error message and allow for data re-entry.

(c) Write a tester program to test out the methods of your `EasyScannerPlus` class.

3. Look back at the time table application that you developed in programming Exercise 8 of Chap. 8. Here is a reminder of the original design for the `TimeTable` class:




As you can see several methods return **`boolean`** values. Some of these **`boolean`** values were sent to indicate errors. Now modify this class to make use of exceptions as follows:

(a) Develop a `TimeTableException` class (make this an unchecked exception).

(b) Modify the `TimeTable` class so that the constructor, and the methods `makeBooking`, and `cancelBooking` all throw a `TimeTableException` if an error occurs (this would mean that the methods `makeBooking` and `cancelBooking` no longer need to return **`boolean`** values).

(c) The `getBooking` method currently returns **`null`** if either the given day or period number passed as parameters are invalid. Re-write this method to return an `Optional` value instead.

(d) Modify the tester that you developed for the time table application to take account of the exceptions and `Optional` value you incorporated in parts (b) and (c) above.

4. Look back at the *Hostel* case study of Chaps. 11 and 12 and make use of exceptions and `Optional` types where appropriate. Amend the `Javadoc` documentation for this application to include information on any exceptions you may have included.