Chapter 8

Exception Handling

In This Chapter

- How to handle exceptions in your code
- How to use `try/catch` blocks
- When to use the `finally` keyword
- How to define your own exceptions
- When to use the `throw` and `throws` keywords

In a perfect world, none of your programs would ever result in runtime errors. You would always create wonderfully bug-free code, and users of your programs would never make a mistake. Let's face it though; this is not a perfect world, and errors do happen. These can include incorrect user input, missing resources such as network connections or files, or even just logic errors that remain in your code. The good news is that the Java language provides an excellent way to handle these unexpected errors called exception handling.

The Method Call Stack

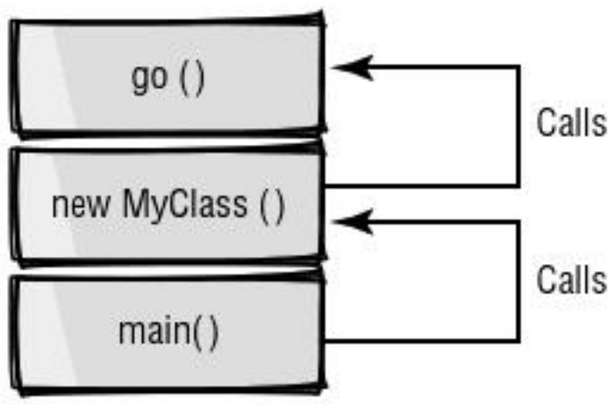
method call stack
All method calls made during runtime are stored on a stack by the JVM. This allows the JVM to maintain information about all the methods that are currently active at a particular moment of runtime. The information in the method call stack is often reported when an exception or error occurs.

Before we venture into all the details of exceptions, it is a good idea to learn some more about how the JVM handles method calls. Although this is not going to be an abundantly technical discussion, I will show you the key concepts of something called the *method call stack*.

Let's begin with the concept of a stack. A stack is a common data structure in programming that represents a last-in-first-out (LIFO) collection of objects. Perhaps the best way to grasp this concept fully is to imagine a stack of dirty dinner plates that you are going to wash. At some point, there was a single plate, then another was placed on top of that one, then another, and so on. In the end, you have a stack of eight plates to wash. Which plate will you wash first? Of course, you're going to grab the plate on top; that only makes sense. This is an example of the LIFO concept. The last plate added to the stack is the first one taken off.

In the Java language, a stack does not hold your dirty dishes but objects. Any number of objects can be stacked on top of each other, and then the topmost object can be removed from the stack when it is needed. When we talk about a stack in this sense, we usually say an object is pushed onto the stack when it is added and popped off of the stack when it is removed.

The JVM uses a stack of objects that hold information about methods during runtime. When you execute a program, you call the `main()` method. The JVM pushes the `main()` method onto the method call stack. If the `main()` method creates an object of type `MyClass` and thus calls the `MyClass` constructor, that constructor becomes the next "method" on the stack. If the `MyClass` constructor calls yet another method named `go()`, that method is pushed onto the stack. The following diagram shows you a conceptual view of the resulting stack as the JVM would see it.



Every time a new method is called, the JVM pushes it onto the stack. Every time a method completes, its information is popped off the stack. By using a stack like this, the JVM knows not only which method is currently being called, but all the method calls that led to it. This ability is important for many reasons; one of those reasons is exception handling.

When a problem occurs during the execution of a particular method, the JVM can report the current information on the stack with something called a *stack trace*. A stack trace allows you to determine the exact process flow at the time of the problem. For example, if a problem was found in the `go()` method described earlier, the stack trace would contain not just the details of the problem, but also a list of every method currently on the stack. You would be able to determine that the problem is in the `go()` method, which was called by the `MyClass` constructor, which in turn was called by the `main()` method. Stack traces can be extremely useful for debugging your code.

That should be enough information about the method call stack to get you rolling in this chapter. As you learn more about exception handling, you will also see more details about how stacks and stack tracing are used.

stack trace
When an exception or error occurs, the JVM can output the current state of the method call stack. This stack trace includes all the classes, objects, and methods currently active and usually includes the line numbers in the source code where the exception occurred. You can print this stack trace using the inherited method `printStackTrace()` located in the `Throwable` class.

Exception Noted

In the previous section, the generic term *problem* was used to describe some unexpected condition in a method call. The actual term used in the Java language for a problem of this sort is an *exception*. An exception is an object that is created to indicate a failure of some kind in a program. This failure could be serious (such as running out of memory) or something easily remedied (such as forgetting to enter a username). In this book, you have already come across a few cases in which you were warned that an exception might occur if you do not provide the correct command-line arguments. For example, take a look at the following simple class file.

exception
An object that is used at runtime to indicate that a strange, incorrect result occurred from a method call. A successful method call returns its declared return type, but an unsuccessful method call returns an exception that is usually handled either within the code or by requesting user input.

```
public class Echo
{
    public static void main(String[] args)
    {
        System.out.println(args[0]);
    }
}
```


Here, the `Echo` class compiles without any problems, but if you fail to pass an argument on the command line, upon execution you will receive a message indicating that an `ArrayIndexOutOfBoundsException` has occurred. This output will look a lot like the following:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at Echo.main(Echo.java:12)
```

This message is an example of a stack trace that indicates a single method existing on the stack. The stack trace tells you that an exception was found in the "main thread," which simply means the process in which your code was running. This is followed by the specific exception name, `ArrayIndexOutOfBoundsException`, which includes the index that is "out of bounds." The value "0" indicates that the exception occurred when you tried to access the first element of the array. That makes sense because you provided no first element in the example. The final portion of the message indicates where in the source code the exception was found.

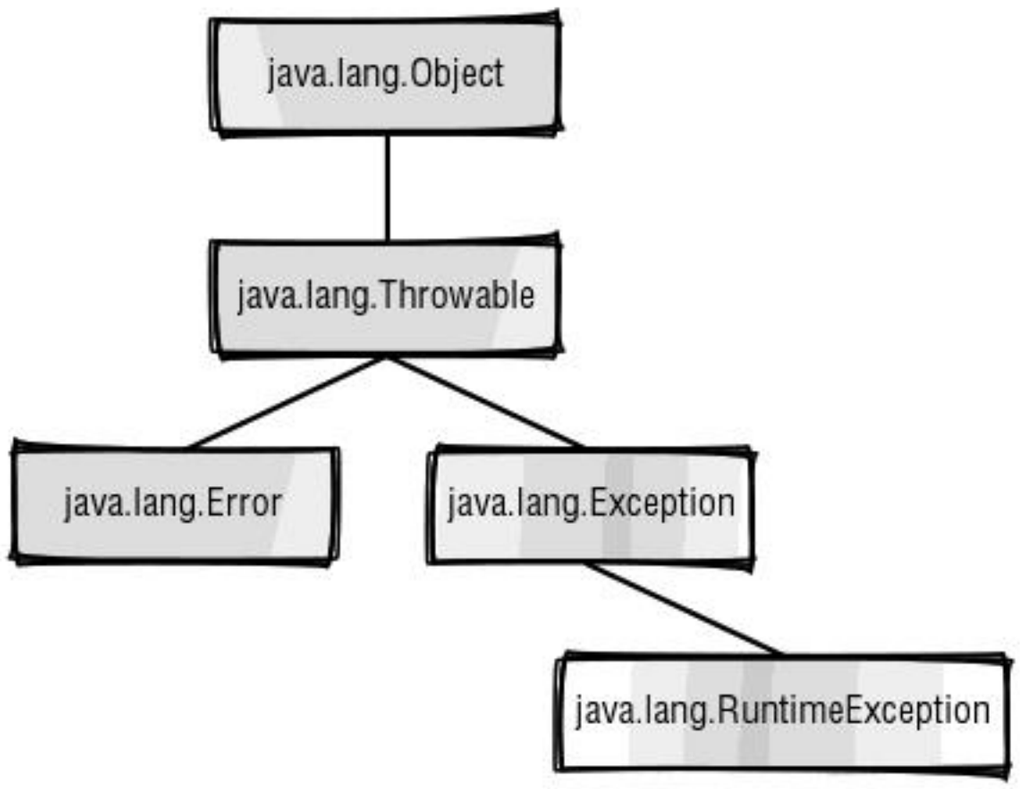
Now, as you may recall, you have already seen how to prevent this particular exception from happening in the first place. You add an `if` statement that checks the length of the array and then outputs a friendly message informing the user how to execute this code correctly. Whenever possible, ensure that your code does everything it can to prevent exceptions in the first place.

However, in some situations, outside forces (such as the user of your program) may inadvertently cause an exception. A critical exception-handling concept is not only recognizing exceptions, but also deciding what happens *after* they are found. This is where the exception-handling mechanism becomes so useful. You can create robust code that "traps" exceptions as they happen and then handles them any way you see fit. You might prompt the user for input, perform some processing of your own that attempts to correct the problem, or even log the exceptions somewhere. The point is that the exception-handling mechanism puts the control where it belongs—in the developer's hands.

Exceptions are bound to methods only. Any method can have a special clause added to it that indicates the potential exceptions that can occur inside the method body. This includes methods defined in the standard application programming interface (API) as well as your own methods. When a method includes an exception clause, the caller of that method is forced to handle the exception somehow. This is what exception handling really means; you will learn all the ins and outs of this syntax as you forge ahead through this chapter.

The Exception Hierarchy

All exceptions are defined as classes. Four classes define the generic exception types available in Java programs; all these classes are found in the `java.lang` package, and they form the core hierarchy for all exception types. Of course, the highest class in the hierarchy is `java.lang.Object`, so the following diagram actually shows five classes.



The remaining classes in this hierarchy are discussed next.

java.lang.Throwable This class contains methods that all exception types will share. The most commonly used method defined in this class is `printStackTrace()`, which prints the exception information to the *standard error*. Another method defined in this class is `getMessage()`, which returns a `String` object that often contains a more "user-friendly" message than the standard stack trace. This class defines two constructors. The first takes no arguments, and the second takes a `String` parameter. The second constructor allows you to set whatever message you want to be returned from the `getMessage()` method.

standard error
The location where error messages are output. Typically, this is the same as the standard output where normal messages are printed. Sometimes the standard error of a system is a log file or a printer somewhere. Whenever you call the `System.err.println()` method, the provided message will typically be sent to the standard output of the system on which the code is running.

java.lang.Error This class (and any subclass) indicates a serious *error* condition that you really cannot do anything about from within your program. This is usually some internal JVM problem. An example of an `Error` subclass is the `java.lang.OutOfMemoryError`, which occurs when no more memory is available for a program to use. Obviously, if you run out of memory, there is not much you can do about it! Normally, when an `Error` is created, the program terminates completely.

error
An object that indicates that a severe condition has arisen in the JVM. An error is not something that can usually be corrected on the fly at runtime, such as running out of memory.

java.lang.Exception You will primarily focus on this class in your own exception handling. This class or subclasses of it indicate logical problems found during code execution. In many cases, your code can recover from an exception either through user intervention or with logic that you include within the code itself. The standard API includes a host of `Exception` subclasses. The occurrence of such an exception does not necessarily mean the program can no longer execute; this is not necessarily a critical problem. As you continue through this chapter, you will learn how to handle exceptions of this type.

checked exception
An exception that the compiler forces code to handle. You can usually accomplish this by using a `try/catch` block. All checked exceptions inherit from the `java.lang.Exception` class but do not extend the `java.lang.RuntimeException` class.

Such classes are considered *checked exceptions*. This means that you are forced to address them (and typically attempt to correct them!) right in the code. This approach ensures that your code does not let potential problems slip by. These exceptions are called checked exceptions because the compiler "checks" to be sure that you are handling them appropriately in your code.

Checked exceptions indicate that some logical, recoverable failure has occurred during program execution. For example, if a username is required before a program can be run, an exception might be generated that triggers a prompt to the user to enter the username.

How does the compiler perform this trick? It simply analyzes any method that you call from your code, and if that method indicates that a checked exception could occur, the compiler informs you that you have to deal with it. How you actually deal with these potential exceptions is discussed in the next section, "Handling Those Exceptions."

unchecked exception
Any exception that extends from `RuntimeException` is considered an unchecked exception. Unchecked exceptions do not have to be handled in a `catch` block, though they can be. The compiler never forces you to handle unchecked exceptions.

java.lang.RuntimeException This class is a special subclass of `Exception` that indicates an *unchecked exception*. An unchecked exception does not have to be directly handled in your code, though it certainly can be. Unlike the way it handles checked exceptions, the compiler does not ensure that you are handling these potential exceptions in your code. This type of exception still indicates a logical error that you could potentially handle. An example of this type of exception is the `java.lang.NullPointerException`, which occurs if you attempt to access a member of an object that has not yet been instantiated. Because this could happen almost anywhere in your code, it would be messy to handle a potential `NullPointerException` inside every single method.

Runtime exceptions typically indicate a problem at the source-code level itself. For example, although it is possible that a `NullPointerException` could happen anywhere that an object reference is used, well-written, robust code ensures that this condition never arises.

Handling Those Exceptions

All right, let's get down to it, shall we? You now know that all exceptions are actually objects, but now it is time to learn how to handle them in your code. The first thing we will concentrate on is how to handle some of the standard exceptions defined in the Java API. Once you have mastered these procedures, you will learn how to create your own exception types.

You provide exception handling by using the keywords `try` and `catch` to form `try/catch` blocks. In some cases, you can also use another keyword, `finally`. This section starts with the `try` and `catch` keywords to get you familiar with the most common approach to exception handling and then it adds the `finally` keyword to the mix to complete the discussion.

Using *try* and *catch*

The `try` and `catch` keywords are aptly named. The `try` keyword is used to form a `try` block. It is inside this block that you call any methods that *could* result in an exception. In other words, a `try` block tells the JVM, "Try to execute these statements, and if an exception happens, let me know." The code inside a `try` block is called the *protected code* because the JVM checks any code within such a block at runtime to determine if an exception occurs. Each `try` block usually has one or more `catch` blocks associated with it. As the name suggests, a `catch` block is used to "catch" an exception that was "thrown" in the corresponding `try` block. In other words, if a statement in the `try` block results in an exception, the code defined in a `catch` block is triggered. A `catch` block is how the JVM lets you know about any generated exceptions.

protected code

The code inside a `try` block is considered "protected" because the JVM is monitoring the results of each statement. If an exception occurs, the JVM attempts to find a matching `catch` block that can process the exception appropriately.

The general syntax of a `try/catch` block is shown here.

```
try
{
    // method calls
}
catch(ExceptionClass name)
{
    // code to execute when this exception type is thrown
}
```

Notice that an exception type wrapped in parentheses immediately follows the `catch` keyword. This can be called the "`catch` parameter" because it acts much like a parameter passed to a method. Technically, the class specified as the parameter must be a subclass of `java.lang.Throwable`. However, more often than not, it is a direct subclass of `java.lang.Exception`. Although this exception can certainly be of the unchecked variety (that is, a class that extends from `RuntimeException`), it is more common to catch checked exceptions.

Don't forget that the braces ({ }) indicate scope for both the `try` and `catch` blocks. Any variable declared inside a `try` or `catch` block can be accessed only from within that same block of code.

WARNING

The `catch` parameter can be referred to anywhere within the scope of the `catch` block itself. This parameter is a reference to an object of the associated exception type, so any `public` methods defined or inherited by this class type can be called. The two most common methods called are `printStackTrace()` and `getMessage()`, both of which are inherited from `java.lang.Throwable`.

The process works quite simply. When a method call contained in a `try` block results in an exception, the JVM takes over the processing and creates a new instance of the particular exception type. The JVM then looks for a matching

`catch` block associated with the `try` block. To be considered a match, a `catch` block must have a parameter that is the same type as the generated exception object or one of the exception's superclasses.

Let's look at more complete example. Earlier in this book, you used the `Integer.parseInt()` method to extract a primitive `int` value from within a `String` object. At that time, you were told that if the characters in the `String` object did not actually form a valid integer value, you would receive a `java.lang.NumberFormatException`. Although `NumberFormatException` is really a runtime (and therefore unchecked) exception, it is not uncommon for you to provide code as shown here to handle invalid input.

```
1  public class SquareIt
2  {
3      public static void main(String[] args)
4      {
5          if(args.length != 1)
6          {
7              System.err.println("Usage: java SquareIt
                   ➥<number>");
8              System.exit(1);
9          }
10
11         int num;
12         try
13         {
14             num = Integer.parseInt(args[0]);
15         }
16         catch(NumberFormatException nfe)
17         {
18             num = 1;
19         }
20
21         System.out.println(num + " squared is " + (num *
               ➥num));
22     }
23 }
```


The first thing this code does is ensure that an argument was passed on the command line. Note that this is done to avoid any possibility of an `ArrayIndexOutOfBoundsException`. Avoid potential exceptions whenever possible because the exception-handling mechanism does add some overhead to code processing. In simple terms, any code inside a `try` block executes more slowly than code not contained in a `try` block because code in a `try` block has to be

closely monitored by the JVM so that any resulting exceptions can be passed to a corresponding `catch` block.

Do not use exception handling just because you can. Often, you may find it more efficient to perform some tests of your own instead of always using exceptions. Really, you should use exception handling only to handle conditions that you cannot effectively control within the code.

WARNING

Lines 12–19 form the `try/catch` block that checks to be sure that the supplied argument is a valid integer. If it is, the code in the `try` block executes normally, and the resulting parsed value is stored in the *num* variable. If the argument value is not a valid integer value, a `NumberFormatException` is created and passed to the `catch` block on line 16. The body of this block simply assigns the literal value 1 to the *num* variable. The end result is that whether an exception occurs or not, *num* has a valid value.

Let's go back to the case where an exception did not occur. Obviously, in such a case, you do not need to execute the code contained in the `catch` block. If line 14 works without exception, the next line of code that the interpreter executes is line 21—it prints the squared value of *num*. On the other hand, if line 14 does result in a `NumberFormatException`, the body of the `catch` block executes (line 18), and then the squared result prints (line 21).

The following code shows two examples of executing the `SquareIt` code. The first example executes without exception; the second example causes an exception. However, because you are using a `try/catch` block, the code executes without incident using the value of 1.

```
java SquareIt 10
10 squared is 100
java SquareIt Hello
1 squared is 1
```


How Exceptions Change Process Flow

Now check out the following revised version of the `SquareIt` code. The only difference is an extra printout on line 15 that follows a successful parsing of the supplied argument. This is a key difference, though. If line 14 results in an exception, does line 15 execute?

```
1  public class SquareItAgain
2  {
3      public static void main(String[] args)
4      {
5          if(args.length != 1)
```

```
6         {
7             System.err.println("Usage: java
  ➥SquareItAgain <number>");
8             System.exit(1);
9         }
10
11        int num;
12        try
13        {
14            num = Integer.parseInt(args[0]);
15            System.out.println("No problem!");
16        }
17        catch(NumberFormatException nfe)
18        {
19            num = 1;
20        }
21
22        System.out.println(num + " squared is " + (num *
            ➥num));
23    }
24 }
```

To find the answer, let's run the `SquareItAgain` code. As before, the first execution does not cause an exception, and the second execution does.

```
java SquareItAgain 10
No problem!
10 squared is 100

java SquareItAgain Hello
1 squared is 1
```

As you can see, only the message "No problem!" prints in the first case. This is because of the way processing works in a `try` block. Once an exception occurs, control immediately passes to the `catch` block. Any remaining statements contained in the `try` block are skipped entirely and are not executed again. Instead, the body of the `catch` block executes (line 19, which sets the value of `num` to 1), and then the next line following the `catch` block executes (line 22, which prints the result).

Adding More *catch* Blocks

Although a `try/catch` block will have only one `try` statement, you can have multiple `catch` blocks. This is an important concept because it is possible that a `try`

block can throw more than one exception. You will see a realistic example of this when you learn how to create your own exception types in the "Creating Your Own Exception Type" section later in this chapter, but the general syntax is shown here. Note that the method and exception class names are examples only.

```
try
{
    methodA(); // might generate OneException
    methodB(); // might generate AnotherException
}
catch(OneException oe)
{
    // handle OneException
}
catch(AnotherException ae)
{
    // handle AnotherException
}
```

This pseudocode has two `catch` blocks associated with the `try` statement. If the call to `methodA()` results in the `OneException`, the first `catch` block executes. If `methodB()` results in the `AnotherException`, the second `catch` block executes. Only *one* `catch` block will ever execute when you run this portion of code. Multiple `catch` blocks cannot execute in this example because only one exception can ever be active. As soon as an exception occurs, that is the end of all the "normal" processing; the exception-handling mechanism kicks in and immediately passes control to the appropriate `catch` block.

Ordering the Exceptions Correctly

The first `catch` block that matches a given exception type is the only one that executes. Because both checked and unchecked exceptions extend from the generic `Exception` class, you have to consider the order in which you place your `catch` blocks.

Take a look at the following code snippet. Which `catch` block executes if a `OneException` is generated by the call to `methodA()`?

```
try
{
    methodA();
}
catch(Exception ex)
{
    System.err.println(ex.getMessage());
```

```
}
catch(OneException oe)
{
    System.err.println(oe.getMessage());
}
```

In this example, the first `catch` block *always* executes. In fact, regardless of the actual exception type, the first `catch` block is the only block to execute. This is because the generated exception object is a subclass of the `Exception` class. In Chapter 7, "Advanced Object-Oriented Programming," you learned that an object type is either its actual class type or any of its superclasses. So, because the initial `catch` block traps any class that subclasses `Exception`, it is the only block to execute.

It is not uncommon to have multiple `catch` blocks in your code like this, so you have to remember to order those blocks appropriately. If you change the code as shown in the following example, the code will trap any `OneException` objects in the first `catch` block and any other exceptions in the second.

```
try
{
    methodA();
}
catch(OneException oe)
{
    System.err.println(oe.getMessage());
}
catch(Exception ex)
{
    // catches everything except OneException
    System.err.println(ex.getMessage());
}
```

Always order your `catch` blocks from the most specific to the most generic to ensure that the correct blocks execute when they should. Remember, only one `catch` block ever executes, and it is the one that first matches a given exception type. In the preceding code, if `methodA()` results in a `OneException`, the first `catch` block executes. If `methodA()` throws an `ArrayIndexOutOfBoundsException`, the second `catch` block executes instead.

Using a *finally* Clause

As mentioned at the start of this section, a third keyword can be used in relation to exception handling. If you need to ensure some processing, whether there was

an exception or not, you will find a `finally` clause useful. Most often, you use a `finally` clause to free some external resource. For example, you might open a file and then attempt to write a few lines of data to the file. This newly written data could work fine, but it could also cause an exception. A `finally` clause can close the file regardless of whether the new code worked correctly.

Of course, you have not learned how to open and write to files, so it would do you little good to show such code now. Instead, the following `SquareIt-Finally` code demonstrates how the `finally` clause works.

```
1  public class SquareItFinally
2  {
3      public static void main(String[] args)
4      {
5          if(args.length != 1)
6          {
7              System.err.println("Usage: java
   ➥SquareItFinally <number>");
8              System.exit(1);
9          }
10
11         int num;
12         try
13         {
14             num = Integer.parseInt(args[0]);
15             System.out.println("No problem!");
16         }
17         catch(NumberFormatException nfe)
18         {
19             num = 1;
20         }
21         finally
22         {
23           System.out.println("This always prints out.");
24         }
25
26         System.out.println(num + " squared is " + (num *
              ➥num));
27     }
28 }
```

If you compile and execute the `SquareItFinally` code just as you did in the previous two examples, you will see the following output. Notice that the printout inside the `finally` block always executes.

```
java SquareItFinally 10
No problem!
This always prints out.
10 squared is 100

java SquareItFinally Hello
This always prints out.
1 squared is 1
```


As you can see, using `finally` is fairly straightforward. However, there is something special about a `finally` clause—it *always* executes, no matter what. Look at the following code snippet from the `SquareItFinally` code; I have altered it to show you an example of what I mean by "always."

```
try
{
    num = Integer.parseInt(args[0]);
    return;
}
catch(NumberFormatException nfe)
{
    num = 1;
}
finally
{
    System.out.println("This always (and I mean always)
prints out!");
}
```

If you add that `return` statement to the code, you might expect that if the `Integer.parseInt()` method works successfully, processing immediately returns to the calling method. After all, that is what `return` does, right? Well, normally, I would say yes, but not this time. In this case, what actually happens is that the integer is parsed, the message in the `finally` clause is printed, and *then* control is returned to the caller of this code. The same concept applies to `break` and `continue` as well.

Bypassing a `return` statement as described here is not common, but you should be sure that you understand the implications of using `finally`, or you could be quite surprised at the results! The only way that a `finally` block will not execute is if a call to `System.exit()` is made; this ends all processing altogether.

NOTE

Creating Your Own Exception Type

Although the Java API provides many exception classes, you can also create your own exceptions. You can create various types of exceptions that indicate specific problems in your code. Luckily for us, creating exceptions is a simple process. All you usually have to do is subclass the `Exception` class and perhaps override a single method.

To see how to create a custom exception type, you will define a new class named `NoSuchEntryException`, which you will then use with a personal phone book application that I introduce in the next section. The concept is that if you search for a particular entry in the phone book and you cannot find it, this exception indicates the failure.

This section defines only the `NoSuchEntryException` class. In the next section, where you learn how to use `throw` and `throws`, you actually use this exception in a complete program.

NOTE

Usually when you define your own exception type, you want it to be a checked exception so that the compiler can ensure that you are handling the exception appropriately. To create a new checked exception, simply subclass the `Exception` class. If you think about it, you are just adding a new exception class to the exception hierarchy that we discussed earlier in this chapter in the section "The Exception Hierarchy."

Whenever you create your own exceptions, you follow some common steps. Although you can certainly add your own methods and variables, it is more common to simply use the inherited methods defined in your superclasses. First, you want to define at least two constructors: a default, no-argument constructor, and a constructor that accepts a `String` object as a parameter. The latter constructor allows you to pass the message that you want this exception type to return from the `getMessage()` method. Normally, all these constructors do is invoke the superclass constructors, located in `Exception`.

The `Exception` class also provides these two forms of constructors; they simply invoke the constructors in its superclass, `Throwable`.

NOTE

Here is the initial version of the `NoSuchElement` class:

```
public class NoSuchEntryException extends Exception
{
```

```
    public NoSuchEntryException()
    {
        super();
    }

    public NoSuchEntryException(String message)
    {
        super(message);
    }
}
```

Hey, I wasn't kidding when I said it was simple! Remember that your class inherits the key methods `getMessage()` and `printStackTrace()` from the `Throwable` class, so you do not need to redefine them unless you want to add some new functionality.

However, you might often want to augment the existing functionality with your own information. Let's expand this class to make it a bit more useful. Because we are going to use it with a phone book application to indicate a failed search, it might make a lot of sense to include the information being searched for in the output message. To make this happen, you simply override the `getMessage()` method, add a third constructor, and define an instance variable.

Here is the final version of the `NoSuchEntryException` that includes these modifications:

```
public class NoSuchEntryException extends Exception
{
    private String searchDetails;

    public NoSuchEntryException()
    {
        super();
    }

    public NoSuchEntryException(String message)
    {
        this(message, "");
    }

    public NoSuchEntryException(String message, String
searchDetails)
    {
        super(message);
```

```
        this.searchDetails = searchDetails;
    }

    public String getMessage()
    {
        String msg = super.getMessage();
        msg += "\nNo match for: " + searchDetails;
        return msg;
    }
}
```

The third constructor allows you to pass the details of the search as a single `String` object. Notice that the `message` parameter is still passed to the superclass, but the new `searchDetails` parameter is completely contained within the `NoSuchEntryException` class. In this code, the `getMessage()` method has also been overridden. First, this method calls the `getMessage()` method of the superclass, and then it appends a message that indicates the search details that failed.

Be sure to notice that no "new" methods are defined in the `NoSuchEntryException` class. The `getMessage()` method is a standard exception method; you are simply overriding it in this class. This means that users of this exception do not have to learn any new methods; they simply rely on the standard methods they expect to find. This is another example of polymorphism, as discussed in Chapter 6, "Introduction to Object-Oriented Programming," and Chapter 7, "Advanced Object-Oriented Programming."

Now that you have created the exception, go ahead and compile it to be sure that you have no syntax errors. In the next section, you will use this exception in a complete program.

TIP **You can extend from the generic `Exception` class or any of its subclasses. If an exception class is already defined and you simply want to expand on it, inherit directly from this class instead.**

Throwing Exceptions

Up to now, you have learned how to catch any exceptions and how to create your own exception classes. Now it is time to actually throw these exceptions, the final piece of the exception puzzle.

Throwing exceptions involves two aspects. The first is defining your methods so they indicate that they might generate an exception. The second is including logic to determine if an exception should occur and then actually creating it and "throwing" it from within the method.

Using the *throws* Keyword

throws clause
Any method declaration can include the `throws` keyword followed by a list of one or more exceptions that might result from calls to the method. If a method throws any checked exceptions within its body, those exceptions must be listed in the `throws` clause.

If you define a method that can generate an exception, you use the `throws` keyword followed by the list of possible exceptions. This can be referred to as a *throws clause*. You place the `throws` clause immediately after the parameter list, as shown here:

```
public void methodA() throws OneException{…}
```

This method declaration adds a `throws` clause that indicates it is possible that a `OneException` could occur as a result of calling this method.

NOTE

Adding a `throws` clause indicates only that a method could generate an exception, not that it actually will. In fact, most of the time, one would hope that no exception occurs.

If you want to throw a checked exception from a method, you must provide a `throws` clause with that exception (or a superclass of it) in the list of exceptions. If a method can throw more than one exception, separate each exception type with a comma. This example shows a method that can throw two exceptions.

```
public void methodB() throws OneException,
AnotherException{…}
```

NOTE

You have to list only the checked exceptions that you might throw from within a method body. Although you can list unchecked exceptions (such as `NullPointerException`), doing so is an uncommon practice. You can throw all unchecked, runtime exceptions from any method without listing those exceptions in the `throws` clause.

The "Declare or Handle" Rule

One result of declaring methods with a `throws` clause that contains checked exceptions is that it forces any code calling this method to deal with the potential exception. The calling method can deal with an exception in two ways: by "handling" the exception, or by "declaring" it.

You handle an exception by providing a `try/catch` block, as you have already seen. This is the most common approach. Declaring an exception is the other option. In this case, you do not provide a `try/catch` block; instead, you add a `throws` clause that lists the same exception type (or types) to the calling method. For example, the following example shows how the "declare" portion of the "declare or handle" rule works.

```
public void testMethod() throws OneException
{
    methodA():
}
```

Because no `try/catch` block is provided, `testMethod()` provides a `throws` clause of its own. The end result is that the caller of `testMethod()` has to either "declare" or "handle" the exception. Normally, one of the methods that ends up in the call stack provides a `try/catch` block.

The bottom line is that if you write code that calls a method with a `throws` clause that includes checked exceptions, the compiler forces you to either declare or handle that exception. This eliminates the possibility of creating code that is ignorant of potential runtime problems.

NOTE

Remember, only checked exceptions must follow the "declare or handle" rule. You should normally also deal with any unchecked exceptions, of course, but the compiler does not enforce this behavior. If an unchecked exception occurs and is never caught, your program will crash and dump the stack trace to the command prompt.

Because you cannot really write a complete version of the phone book code until you read the next section, you will create only an interface at this point. In Chapter 7, you learned that an interface defines all the methods that any implementing class must override. You will implement this interface in the next section.

```
public interface PhoneBook
{
    public void addEntry(String first, String last, String
number);
    public String search(String criteria) throws
NoSuchEntryException;
}
```

The `PhoneBook` interface defines two methods: The `addEntry()` method allows new entries to be added to the phone book; you can use the `search()` method to search the entries based on the criteria provided. The `search()` method also specifies that it can throw a `NoSuchEntryException`.

Throwing Exceptions in a Subclass

If you override a method and that method includes a `throws` clause, you have to follow a simple rule. The overridden version of the method can only throw the same exception types or subclasses of those exceptions as defined in the superclass version of the method. You cannot throw any other exception from an overridden method.

If you recall the discussion about overriding methods and polymorphism, this rule will make complete sense. If a subclass method could throw different exceptions than the overridden superclass method, the JVM could get confused. If you accessed the method via the parent class, that method could generate some set of exceptions. If you accessed the method via the subclass, an entirely different set

of exceptions could be thrown. Because this causes confusion, it is a good thing that you simply cannot break this rule!

An overridden method can, however, throw *fewer* exceptions than the method defined in the superclass. The only thing you cannot do is expand the list of exceptions thrown by the overridden method.

The *throw* Keyword

The second aspect of throwing exceptions is to actually do the throwing, of course. You accomplish this by creating a new instance of the exception class, and then you use the `throw` keyword. Think of the `throw` keyword as a special form of the `return` statement that returns only exceptions. Once a `throw` statement executes, the method stops executing immediately, and the exception object is passed back to the previous method in the call stack. As you just learned, the method receiving the thrown exception object must now either declare it or handle it.

WARNING

Don't confuse the `throw` and `throws` keywords. The `throws` keyword is used right in the method declaration, whereas the `throw` method is used within the method body.

You can use a conditional expression, usually an `if` statement, to determine if an exception should be thrown. Here is a pseudocode example of this process:

```
public void methodA() throws OneException
{
    boolean okay = checkSomeCondition();
    if(okay)
    {
        // do whatever processing you like
    }
    else
    {
        // condition failed, throw the exception
        throw new OneException("Oops!");
    }
}
```

Of course, the `checkSomeCondition()` method call in this code is completely made up, but assume that it performs some test and returns a `boolean` result. If the result is `true`, everything is working fine, and whatever processing that needs to be done can continue. However, if the result is `false`, an exception should be thrown. Notice that an object of the required exception type follows the `throw` keyword. Once this statement executes, `methodA()` terminates, and control is immediately passed to the next method in the call stack.

The Complete Phone Book Application

Okay, you have now seen examples of everything you need to write a complete class that generates exceptions and can handle them appropriately. It is time to put together all you have learned in the final phone book application. The phone book application is a little more involved than many of the other examples that you have seen thus far, but if you spend some time reading the code and the descriptions that follow each class, it should be clear.

The first class you will need is the `Entry` class, which holds the details of a phone book entry. This class is defined as follows:

```
public class Entry
{
    private String first;
    private String last;
    private String number;

    public Entry(String first, String last, String number)
    {
        this.first = first;
        this.last = last;
        this.number = number;
    }

    public String getFirst()
    {
        return first;
    }

    public String getLast()
    {
        return last;
    }

    public String getNumber()
    {
        return number;
    }

    public String getDetails()
    {
        return first + " " + last + " " + number;
    }
}
```

The `Entry` class contains three instance variables that hold the first name, last name, and phone number. The constructor sets the values of these three variables. The first three methods are accessor methods, one for each variable. The final `getDetails()` method is a convenience method that returns all the values in a single `String` object.

Next, the `MyPhoneBook` class implements the `PhoneBook` interface that you have already created. Both of the methods in the `PhoneBook` interface are overridden, and a simple `String` array is used to store the elements. Note that because an array must have a consistent length, a length is specified in the constructor.

NOTE

It would be a lot more useful if you could store the phone book entries in a structure that could actually change its size rather than in a fixed-size array. In Chapter 9, "Common Java API Classes," you will learn about some different types that allow you to create those kinds of structures. For now, you will stick to using a simple array so that the code remains as clear as possible.

```
public class MyPhoneBook implements PhoneBook
{
    private Entry [] entries;
    private int index;

    public MyPhoneBook()
    {
        // default to a size of 10
        this(10);
    }

    public MyPhoneBook(int size)
    {
      entries = new Entry[size];
}

public void addEntry(String first, String last, String
   ➥number)
{
    Entry entry = new Entry(first, last, number);
    try
    {
        entries[index++] = entry;
    }
```

```
        catch(ArrayIndexOutOfBoundsException ae)
        {
            System.err.println("Unable to add entry.");
            System.err.println("The phone book is full.");
        }
    }

    public String search(String criteria) throws
       ➥NoSuchEntryException
    {
        String details = "";
        for(int i = 0; i < entries.length; i++)
        {
            Entry entry = entries[i];
            if(entry.getFirst().equalsIgnoreCase(criteria) ||
    ➥entry.getLast().equalsIgnoreCase(criteria) ||
    ➥entry.getNumber().equals(criteria))
            {
                details += entry.getDetails() + "\n";
            }
        }

        if(details.equals(""))        {
            throw new NoSuchEntryException("No entry found",
               ➥criteria);
        }
        else
        {
            return details;        }
    }
}
```

Here I have used the constructors to initialize the array of `Entry` objects, either to the default size of 10 or to the value passed into the second constructor. The `addEntry()` method attempts to add a new `Entry` object using the parameter values passed to it. Notice that in this method, I have used a `try/catch` block to handle an `ArrayIndexOutOfBoundsException`. In addition, I have used the instance variable *`index`* to add a new `Entry` to the array, and then I have incremented the *`index`* variable by one. So, if you attempt to add an `Entry` object to the array beyond the fixed size, the `catch` block will trap the `ArrayIndexOutOfBoundsException`.

NOTE

The logic of the addEntry() method could be improved a bit. For simplicity, I have not made any checks to ensure that the three parameters actually hold logical values. As a result, once you have completed this code, you might want to tinker with it a bit to validate the input values.

The `search()` method iterates through the `entries` array, checking to see if any of the variables match the given criteria. I achieved this using the "Boolean OR" (`||`) operator. You will also notice that instead of the standard `equals()` method, I used the `equalsIgnoreCase()` method to test the first and last name. The `equalsIgnoreCase()` method is defined in the `String` class so that you can compare the two `String` objects without worrying about case sensitivity.

If a match is found by the `search()` method, the details of the `Entry` object are appended to the *`details`* variable. Once the entire array has been checked, the method reaches the `if` statement that performs the conditional test to see if any entries were found. If the *`details`* variable still has its default value of `""`, no matches were found, and the exception should be triggered. You can accomplish this by using the `throws` keyword followed by a new `NoSuchEntryException` object. If the *`details`* variable holds an actual value, indicating that some matches were found, the *`details`* variable is returned.

The last class that you need is one that uses a `MyPhoneBook` object and allows you to search for entries. The `TestPhoneBook` class that follows expects you to pass an argument that it will use to search the created phone book.

```
public class TestPhoneBook
{
    public static void main(String[] args)
    {
        if(args.length != 1)
        {
            System.err.println
➥("Usage: java TestPhoneBook <criteria>");
            System.exit(1);
        }

MyPhoneBook book = new MyPhoneBook(4);
book.addEntry("Joe", "Smith", "999-555-1000");
book.addEntry("Fred", "Jones", "888-555-2000");
book.addEntry("Sally", "Cortez", "777-555-3000");
book.addEntry("Nancy", "Smith", "444-555-4000");

try
{
    System.out.println(book.search(args[0]));
```

```
            }
            catch(NoSuchEntryException ne)
            {
                ne.printStackTrace();
            }
        }
    }
```

After ensuring that you passed an argument, the `main()` method creates a new `MyPhoneBook` object and adds four entries. Finally, the code searches the phone book using the argument you provided. Because the `search()` method is defined to throw a `NoSuchEntryException`, a `try/catch` block handles this potential exception. If the call to `book.search()` works without a problem, the entries that were found in the phone book are printed. However, if no matching entries were found, a `NoSuchEntryException` is passed to the `catch` block.

The logical thing to do with the `catch` block is call the standard exception method, `printStackTrace()`. This outputs a detailed message about the exception that occurred. It first includes the output of the `getMessage()` method, and then it includes the methods currently in the call stack, usually with the line number where the error occurred.

Let's run this code and see what happens. First, it searches for an entry that exists so that you can see the code work correctly. After all, code that works correctly is usually the idea!

```
java TestPhoneBook Smith
```

The output from this execution is as follows:

```
Joe Smith 999-555-1000
Nancy Smith 444-555-4000
```


Great, everything seems to be working wonderfully. Now it is time to intentionally search for an entry that does not exist.

```
java TestPhoneBook Williams
```

This time the output is the stack trace of the thrown exception:

```
NoSuchEntryException: No entry found
No match for: Williams
    at MyPhoneBook.search(MyPhoneBook.java:54)
    at PhoneBookTest.main(PhoneBookTest.java:26)
```

The first two lines are the results of calling `getMessage()`. The last two lines contain the stack trace information. The first of these lines indicates the current method in the call stack, which will always be the last method where

the exception was thrown. In this case, that is the `MyPhoneBook.search` method. It includes (in parentheses) the source file and the line number where the exception occurred. If you look at the line number that the exception indicates in the `MyPhoneBook` source file, you will see that it is the following line:

```
throw new NoSuchEntryException("No entry found", criteria);
```

Well, that sure makes a lot of sense, doesn't it? That is definitely the line where the exception was thrown. Normally, it is the first method listed in the stack trace that is of most concern to you for debugging purposes.

The last line indicates the `main()` method, which is the last method in the call stack. A line number is given for this method as well:

```
System.out.println(book.search(criteria));
```

Again, this makes sense because it is the call to the `search` method that generated the exception in the first place.

NOTE

Sometimes the output of a stack trace will not have the source file and line number in the parentheses. Instead, it will say "(Compiled code)". This happens if no source file is available.

So what have you learned with this phone book application? You now know how to define your own exception type, which you did with the `NoSuchEntryException` class. You also learned how to test for an exceptional condition and throw an exception accordingly by implementing the `search()` method in the `MyPhoneBook` class. In the `TestPhoneBook` class, you provided a `try/catch` block to handle a possible exception. Finally, you learned how to analyze a stack trace to track down a problem in the source code so you can fix it.

Terms to Know

checked exception
error
exception
method call stack
protected code
stack trace
standard error
`throws` clause
unchecked exception

Review Questions

1. What type of exception is a `NullPointerException`?
2. From which class do *all* exception and error types inherit?
3. Which method do exception classes use to output the current method call stack information?
4. If a method call results in a user-defined exception that directly extends an `Exception` named `TestException`, is the following catch block legal?

   ```
   catch(Exception ex){}
   ```

5. Which keyword is used to indicate a method that might result in an exception?
6. Which keyword do you use to ensure that certain code always executes whether an exception occurs or not?
7. True or false: A method in a subclass that overrides a method in a superclass that can throw a `TestException` must also throw the `TestException`.