[PART] III

MODERN PROGRAMMING Techniques

Part III examines the implementation phase of the software life cycle. It introduces programming techniques that are critically important in developing modern software.

Today, all software must be robust, fault tolerant, and able to recover from errors in a planned, orderly way. Chapter 10 introduces the Java exception mechanism, which deals with errors and other exceptional conditions, and introduces the concept of stream-based programming, which allows data to come in from any stream-oriented device, such as files, peripherals, and a network.

Using the `java.lang.Thread` class, Chapter 11 presents the topic of multithreaded code, which enables a single piece of software to deal with multiple requests and allows servers that can simultaneously handle input from multiple clients. Chapter 12 introduces graphical user interfaces and shows how to construct elegant and easy-to-use visual interfaces using the `java.awt` and `javax.swing` toolkits.

Finally, Chapter 13 examines computer networks and shows how to construct "net-centric" software that interacts with remote users across a telecommunications channel using the `Java.net` package.

After reading this material, you will be able to develop fault-tolerant, multithreaded, net-centric software with a visual user interface—features that truly represent modern software development.

[CHAPTER] 10

Exceptions and Streams

10.1 Introduction

After a long day at the office, it is time to head home. You find your car in the parking lot, start it, and begin to drive. About halfway home, you hear a thumping noise. You pull off the road and discover that one of the front tires is flat. Fortunately, you know how to change a flat tire, so you quickly put on the spare, get back in the car, and resume the drive with only a minor delay. Flat tires are rare events, but you were prepared for it. Even though you seldom use a spare tire and jack, they must be there when you need them. Having this equipment in your car and knowing how to use it can prevent a small problem from becoming a catastrophe.

Software systems, like human beings, must deal with rare and unexpected events. An unplanned event that lies outside the normal behavior of a software algorithm is called an **exception**. For example, consider what might happen when a user instructs a spreadsheet program to save the current worksheet on a thumb drive. Most of the time, the user places a usable and correctly formatted thumb drive into a USB port before executing this command. However, a well-designed program must also deal with the possibility that the thumb drive is not present, is not usable, or does not have enough room to store the information. All of these situations represent events that are not part of the normal concerns of the algorithm.

An exception is not simply a special case of the algorithm that should be handled by the program as part of its regular flow of control. For example, consider a program that locates the largest number in an array segment of size M within a larger array of length N. The fact that two or more entries might have the same largest value would not be considered an exception. It would be a case that the program needs to check and handle. However, if M, the size of the segment, is greater than N, the situation may need to be handled as an `ArrayIndexOutOfBoundsException`. Similarly, if we are reading data values into an integer array and encounter a string, such as "three", it may be considered a `NumberFormatException`.

The idea of writing a program to deal with unexpected events is nothing new; programmers have been doing so for years. Consider the task of writing the `pop()` method for the `Stack` interface defined in Figure 10-1. This version of `pop()` is slightly different from the one in Figure 6-26. This `pop()` method removes the top value from the stack and returns it, and deals with the possibility that the stack is empty. Clearly, it is not difficult to determine if the stack is empty, but it is harder to decide what to do if this case arises. One possibility, shown in Figure 10-2, is to print an error message and terminate the program.

```
public interface Stack<T> {
    public void push( T element );
    public T pop();
```

continued

```
    public boolean isEmpty();
    public boolean isFull();
}
```

[FIGURE 10-1] Simple `Stack` interface

```
public T pop() {
    T retVal = null;

    if ( isEmpty() ) {
        System.err.println( "Empty Stack" );
        System.exit( 1 );
    }

    // Set retVal to the value on top of the stack,
    // then remove it from the stack. This code is omitted to
    // focus on the issues of exception handling.

    return retVal;
}
```

[FIGURE 10-2] Handling exceptions using `System.exit()`

Although the version of `pop()` in Figure 10-2 works if the stack is empty, how can the programmer be certain that the appropriate action is to terminate the program in this situation? Think about the consequences if automobiles were designed to turn the engine off whenever a flat tire was detected! Often, not enough information is available at the point in the program where an exception occurs to make an informed decision about the best course of action. The exception may need to be passed to a higher level of the program, where enough information is available to decide how to handle the problem.

You can modify the `pop()` method in several ways to pass information about the exception to the calling method. For example, the code in Figure 10-3 returns a **null** reference to indicate that an exception has occurred.

```
public T pop() {
   T retVal = null;
```

continued

```
    if ( !isEmpty() ) {
        // Set retVal to the value on the top
        // of the stack and remove that element
        // from the stack. This code is not shown.
    }

    return retVal;
}
```

[FIGURE 10-3] Handling exceptions by returning **null**

Although this code passes information about the exception back to the caller of the `pop()` method, the technique still has some serious drawbacks. To use a return value to indicate whether an exception occurred, the value cannot validly be returned by the method as a possible answer. In Figure 10-3, if **null** references could be stored in the stack, the code could not use the value **null** to unambiguously signal that an exception took place. There are ways around this limitation, however. One possibility, illustrated in Figure 10-4, is to define an instance variable within the `Stack` class to indicate whether the last operation was successful. The calling program would check the value of this instance variable after every call to `pop()` to determine if an error occurred.

```
public class Stack<T> {
    // Instance variable to indicate if pop() worked properly
    private boolean success;

    public T pop() {
        T retVal = null;
        if ( !isEmpty() ) {
            // Set retVal to the value on top of the stack and
            // remove it from the stack (code has been omitted).
            success = true;
        }
               else {
                   success = false;
               }

               return retVal;
           }

    public boolean getSuccess() {
        return success;
    }
}
```

[FIGURE 10-4] Handling exceptions using state variables

The major drawback of the technique in Figure 10-4 is that the calling method is not required to check whether an exception has occurred. The `pop()` method can provide information to the caller that an exception occurred, but there is no guarantee that the caller will do anything about it. The coding techniques in Figures 10-3 and 10-4 both depend on the calling program to actually check whether an exception has taken place. The compiler itself does not enforce exception checking and handling.

Many programmers find it tedious to continually check return values or instance variables to determine if a method call was successful. Instead of writing the code shown in Figure 10-5a, programmers often become lazy and assume that the method invocation will work correctly, as shown in Figure 10-5b.

```
val = aStack.pop();
if ( aStack.getSuccess() ) {
    // Handle the successful case
    // (code omitted)
}
else {
    // Handle the error case (code omitted)
}
```

(a) The "correct" way

```
// Assume an exception does not occur
// (i.e., be lazy). Assume val is correct.
val = aStack.pop();
```

(b) The "lazy" way

[FIGURE 10-5] Handling exceptions

A second problem with the approach in Figure 10-5 is that error handling is mixed with the code that implements the logic of the method itself, resulting in programs that can be harder to read and understand. For trivial programs, this may not be an issue, but it can make large-scale programs difficult to work with and maintain. Thus, as we have shown, all the exception-handling approaches described so far have fundamental flaws.

Interestingly, language designers did not begin creating formal control structures that would provide greater support for exceptions until 1975[1]. The basic idea was to provide programmers with an **exception class** (or **type** in a traditional programming language) that could describe unusual or erroneous events. Whenever a method wants to inform a caller that an unusual event has occurred, it creates an **exception object** to describe the event and generates an exception. An **exception handler** in the calling method is invoked to analyze the exception object and take the appropriate actions, which may include passing the information to higher levels of the program.

To incorporate exceptions into your programs, you must understand three basic concepts:

- How exceptions are represented and created. In an object-oriented language, exceptions are usually represented as objects and are handled in the same way as other objects in the language.
- How to define an exception handler to process exceptions that occur in your program. When an exception handler is activated, you need to understand what variables are in scope and what restrictions, if any, are imposed by the language on the code in the handler.
- How control is passed from the program to an exception handler and what happens to the flow of control after the exception has completed.

We address all of these issues in the following sections.

10.2 Exceptions in Java

10.2.1 Representing Exceptions

Exceptions are represented as objects derived from the class `Throwable` or one of its subclasses. The state maintained by a `Throwable` object includes a record of the method calls leading up to the event that caused the exception object and a string that provides text about the exception that occurred. This state can determine where the problem occurred in the program and produce an error message that describes the nature of the problem. Some of the methods defined in the `Throwable` class are listed in Table 10-1.

Java divides the exceptions it recognizes into two broad categories called **unchecked** and **checked**. Unchecked exceptions, which are instances of the `Error` and `RunTimeException` classes, represent serious system errors that a typical program should not try to handle. Examples of unchecked exceptions include running out of memory, stack

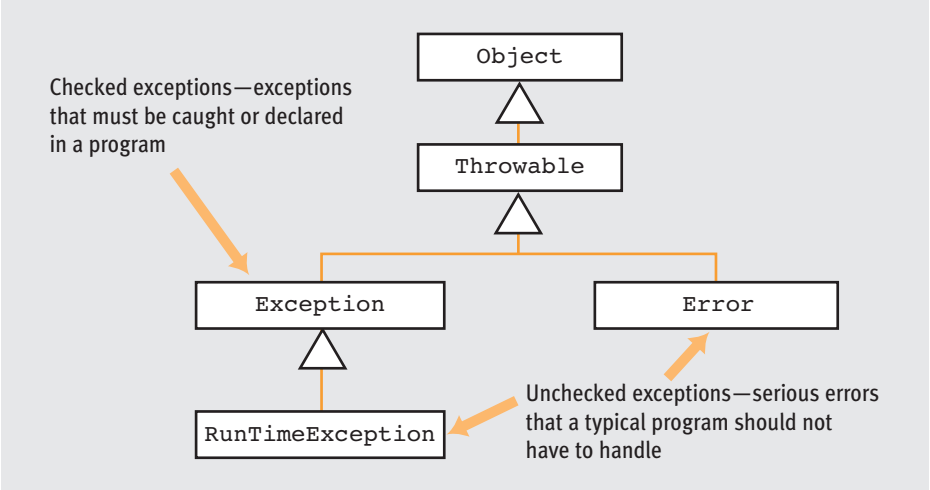
[1] John B. Goodenough, "Exception Handling: Issues and a Proposed Notation, Programming Languages," *Communications of the ACM*, Vol. 18, No. 12, December 1975.

overflow in the Java Virtual Machine, or an attempt to load an invalid class file. All these problems are beyond the user's ability to recover from and continue. Checked exceptions, which are subclasses of the `Exception` class, represent errors that a typical program can handle. An overview of the exception hierarchy is shown in Figure 10-6.

METHOD SIGNATURE	DESCRIPTION
`Throwable()`	Creates a new `Throwable` object with no message
`Throwable(String message)`	Creates a new `Throwable` object with the specified error message
`String getMessage()`	Returns the error message associated with this object

`void printStackTrace()`	Prints a stack trace for this `Throwable` object on the error output stream. A stack trace is a record of the methods that were called and where they may be found in the source code leading up to the event that caused this object to be created. Typical output produced by this method is:
	`java.lang.NullPointerException`
	`  at Stack.pop(Stack.java:20)`
	`  at Stack.main(Stack.java:33)`
	`Exception in thread "main"`

[TABLE 10-1] Methods defined in class `Throwable`



[FIGURE 10-6] Java exception class hierarchy

The Java language specification requires a program to handle or declare any checked exceptions that might be generated, either by providing a handler for the exception (catching it) or informing the calling program that you may generate this type of exception but not handle it (declaring it). This policy, enforced by the Java compiler, is called Java's **catch or declare** policy. It means that a programmer must account for any checked exception that might be generated in a program. Almost all of the exceptions you deal with in a Java program are checked exceptions.

You might wonder why the designers of Java decided to split the exception classes into unchecked and checked categories. If you think about the types of serious run-time errors that unchecked exceptions represent, it is easy to see that the cost of handling unchecked exceptions often exceeds the benefit of catching or specifying them because the user generally cannot recover from them. Thus, the compiler does not require that you deal with unchecked exceptions, although you can if you want. However, checked exceptions represent situations that a program should be able to deal with, and the compiler ensures that these types of exceptions are either caught or declared.

As you might expect, a large number of exception and error classes have already been defined in Java. These exception classes are usually defined within the package that contains the classes that generate the exceptions. For example, all of the exceptions that deal with input and output are in the `java.io` package, and all of the exceptions that deal with networking errors are in the `java.net` package. Table 10-2 lists some of the most common Java exceptions, along with the name of the package in which the exception is defined. By convention, each exception class provides two constructors: a no-argument constructor and a one-parameter constructor that takes a string and provides an error message describing the nature of the exception.

Clearly, the software engineers at Sun could not anticipate every possible exception condition that may occur within a program, so occasionally you may need to write your own exception classes to deal with a special situation. Figure 10-7 shows the specification of a `StackException` class; it could indicate that an error occurred either when attempting to push an item on to a full stack or popping an element off an empty stack. `StackException` is a condition that we expect a program to handle, which is why it extends the `Exception` class of Figure 10-6 rather than the `Error` class.

EXCEPTION	TYPE	PACKAGE	DESCRIPTION
`FileNotFoundException`	Checked	`java.io`	Thrown when a request to open a file fails

continued

EXCEPTION	TYPE	PACKAGE	DESCRIPTION
`IllegalArgumentException`	Unchecked	`java.lang`	Thrown to indicate that a method has been passed an illegal or inappropriate argument
`IOException`	Checked	`java.io`	Thrown to indicate that some sort of I/O error has occurred
`NullPointerException`	Unchecked	`java.lang`	Thrown when attempting to access an object using a **null** reference
`NumberFormatException`	Unchecked	`java.lang`	Thrown when an application attempts to convert a string to a numeric type, but the string does not have the appropriate format
`UnsupportedOperationException`	Unchecked	`java.lang`	Thrown when the object does not provide the requested operation

[TABLE 10-2] Some common Java exceptions

```
public class StackException extends Exception {
  public StackException() {}
  public StackException( String msg ) {
    super( msg );
  }
}
```

[FIGURE 10-7] `StackException` class

Although the `StackException` class in Figure 10-7 is sufficient for describing the two exceptions that typically occur when working with a stack, the only way to determine the exact cause (in other words, whether we attempted to improperly add or remove an item) is to examine the string contained within the exception object.

Consider the two exception classes defined in Figure 10-8. The classes `StackUnderflow` and `StackOverflow` are still `StackExceptions`, but they more accurately define the nature of the problem we encountered. We have created a separate exception class for each of the two types of stack exceptions that can occur. Adding these two classes gives a programmer the flexibility to look for the more general `StackException` or the more specific `StackOverflow` and `StackUnderflow` exceptions. When designing classes that contain methods that generate exceptions, the designer needs to give careful attention to the amount of information the exceptions can provide to the user.

```
public class StackUnderflow extends StackException {
  public StackUnderflow() {}
  public StackUnderflow( String msg ) {
    super( msg );
  }
}

public class StackOverflow extends StackException {
  public StackOverflow() {}
  public StackOverflow( String msg ) {
    super( msg );
  }
}
```

[FIGURE 10-8] Exception classes for use with the `Stack` class

10.2.2 Generating Exceptions

Now that you know how exceptions are represented in Java and how to create them, the next question is what to do with them. Instantiating an exception object is not enough to set the exception-handling mechanism into action. After a program detects that an exceptional situation has occurred and has created an exception object to represent the condition, it must trigger the exception-handling code in the program. The process of invoking an exception handler is called **throwing** an exception. In Java, the `throw` statement is used to raise an exception and to trigger execution of the exception-handling mechanism. The syntax of the `throw` statement is:

```
throw exceptionObject;
```

The `throw` statement takes as an argument a reference to an object from the class `Throwable` or one of its subclasses, as diagrammed in Figure 10-6. The code in Figure 10-9 illustrates the proper use of the `throw` statement.

```
public T pop() throws StackUnderflowException {
  T retVal = null;

  if ( isEmpty() ) {
    throw new StackUnderflowException( "Empty Stack" );
  }

  // Set retVal to the item at the top of the stack,
  // and remove the element from the stack.
  // This code has been omitted.

  return retVal;
}
```

[FIGURE 10-9] Correct use of the `throw` statement

When a `throw` statement is executed, normal execution of the program is terminated and the exception-handling mechanism is put into action. Exceptions that are thrown must be handled, or the program is terminated. Up to this point, you probably have not placed any code in your Java programs to handle exceptions, even though some of the code you wrote might have caused exceptions to be thrown. For example, consider the code in Figure 10-10, which implements a stack using a linked list (refer to Section 6.3.3.2 for more details).

```
// A linked list implementation of a stack. The element at the
// front of the list is the element currently at the top of the
// stack. An item is pushed on the stack by adding it to the
// front of the list. An element is removed from the stack
// by removing the first element in the list.

public class Stack<T> {
    // An inner class used to create the nodes that make up the
    // linked list
    private static class StackNode<T> {
      public T data;              // Data held by this node
      public StackNode<T> next;   // The next node in the list
```

continued

```
    // Create a new stack node
    public StackNode( T d, StackNode<T> n ) {
        data = d;
        next = n;
    }
  }

  // Reference to the node at the top of the stack
  private StackNode<T> tos = null;

  // Push an item on the stack—make it the first node in the list
  public void push( T data ) {
    tos = new StackNode<T>( data, tos );
  }

  // Pop the top of the stack—remove the first element in the list
  public T pop() {
    T retVal = tos.data;  // Get a reference to the data
    tos = tos.next;       // Remove the first node in the list

    return retVal;
  }

  // The stack is empty if the list is empty
  public boolean isEmpty() {
    return tos == null;
  }

  // Create a stack and attempt to remove an element from it
  public static void main( String args[] ) {
    Stack<String> myStack = new Stack<String>();

    // Attempt to pop an empty stack
    System.out.println( myStack.pop() );
  }
}
```

[FIGURE 10-10] Linked list implementation of a stack

Although the program in Figure 10-10 compiles, the message shown in Figure 10-11 appears when the program is executed, and the program terminates.

```
java.lang.NullPointerException
  at Stack.pop(Stack.java:20)
  at Stack.main(Stack.java:33)
Exception in thread "main"
```

[FIGURE 10-11] `Stack` class termination message

The output in Figure 10-11 is an example of a **stack trace** and is the result of calling the method `printStackTrace()` on the exception that was thrown. The first line of the stack trace gives the name of the exception (see Table 10-2) followed by the contents of the call stack at the time the exception occurred. The **call stack** is a record of all the methods that were called up to the point in the program when the exception was thrown. In this example, the method `main()` called the method `pop()`, which caused the `NullPointerException` to be thrown.

An examination of the code in Figure 10-10 reveals the cause of the problem: an attempt to remove an item from an empty stack. Specifically, when the stack was empty, the method `pop()` attempted to execute the following statement:

```
T retVal = tos.data
```

when the instance variable `tos` had the value **null**. Because this was an attempt to access an object using a **null** reference, the Java run-time system threw a `NullPointerException`.

Figure 10-10 never specified how to deal with a `NullPointerException` or that when it occurs, the program should print a stack trace and terminate the program. We did not have to specify how to handle this exception because `NullPointerException` is an unchecked exception. In this case, the Java Virtual Machine handled it by printing a stack trace and terminating the program. However, we could have handled it ourselves, perhaps to implement an alternative response to the specific error.

When an exception is thrown, normal execution of the program stops, and the Java run-time system searches for an exception handler to handle the exception. This search starts at the bottom of the call stack and works its way to the top. The search terminates when either an appropriate handler is found for the exception or the search goes beyond the top of the call stack. An appropriate handler catches either the specific exception that was thrown or any one of its superclasses. For example, a handler that catches a `StackOverflowException`, `StackException`, or an `Exception` in Figure 10-8 could handle the `StackOverflowException`. If a handler is found, the code specified by the handler is executed, and the program continues execution from that point. If a handler is not found, as in Figure 10-11, then when the call stack is exhausted (in other words, all active methods have been searched), the Java Virtual Machine prints a stack trace and terminates the thread in which the exception occurred.

You have not had to worry about exceptions in your program so far because the ones that might have been thrown were all unchecked exceptions. If any other type of exception is thrown in your

program, you must account for it by either providing a handler or declaring that the method may throw the exception. The next section discusses the syntax for writing handlers for exceptions and declaring them.

10.2.3 Handling Exceptions

The first step in writing an exception handler is to enclose all statements that might throw an exception within a **try block**. The general syntax of the `try` statement, which you use to define a try block, is shown in Figure 10-12.

```
try {
    // This is a try statement
}
catch ( ExceptionType e ) {
    // This is a catch block
}
catch ( ExceptionType2 e ) {
    // You can have a catch block for every exception
    // that might occur in the try statement.
}
finally {
    // Executed regardless of what happens in the try block
}
```

[FIGURE 10-12] A `try` statement

You can think of the `try` statement as an "observer" whose job is to monitor the execution of the statements within its try block. As long as no exceptions are thrown inside the try block, the `try` statement does nothing out of the ordinary, and all of its statements are executed exactly as they would be in a regular Java program. However, if any statements inside the try block throw an exception, the `try` statement immediately stops executing the remaining statements in its block. It then examines each of the **catch blocks** that follow the try block, searching for the first exception type parameter that matches the type of exception that was thrown. If a match is found, the code specified inside the catch block is executed, and program execution continues with the first executable statement that follows the `try` statement. If a matching catch block is not found, the method is terminated, the exception is passed up the call chain to the program that invoked this method, and the search process begins again at this level. Figure 10-13 illustrates the use of a `try` statement.

```
try {
    Stack<Integer> myStack = new Stack<Integer>();
    mystack.push( 5 );
    System.out.println( myStack.pop() );
}
catch ( StackUnderflowException e ) {
    System.err.println( "Attempt to pop an empty stack" );
}
catch ( StackOverflowException e ) {
    System.out.println( "Attempt to push an item on a full stack" );
}
```

[FIGURE 10-13] Using a try block to handle exceptions

For example, if the statement `mystack.push( 5 )` is executed when `myStack` is full, a `StackOverflowException` is generated. The `try` statement halts execution of the statements within the try block and searches for a catch block whose parameter matches the `StackOverflowException` or one of its subclasses. A catch block matches the exception that was thrown, so the code within the catch block is executed, causing the message *"Attempt to push an item on a full stack"* to print to standard output. Similarly, if `System.out.println( myStack.pop() )` is executed on an empty stack, the message *"Attempt to pop an empty stack"* prints. The catch block contains the code that executes when an exception occurs. Essentially, it is the exception handler for the exception or any of its subclasses, specified as a parameter to the catch block.

You must follow some simple rules when associating catch blocks with a try block. First, the catch blocks associated with the `try` statement can only specify exceptions that might be thrown by one of the statements in the try block. For example, the `try` statement in Figure 10-13 could not include a catch block that specified a `FileNotFoundException` because this exception cannot be thrown by any of the statements in the try block. Second, the catch blocks must be placed in order from the most specific to the most general exception type. Using the `Stack` class examples, this means that a `StackOverflowException` must be caught before the more general `StackException`, which itself must be caught before the most general `Exception`.

The last aspect of handling exceptions in Java is called a **finally block**. A finally block can be associated with a try block. The finally block is listed after all of the catch blocks and contains code that is guaranteed to execute, regardless of whether an exception occurs within the try block and catch blocks are executed. Finally blocks are used to include code that must execute whether an exception occurs or not.

The most common use of a finally block is to properly close a file. For example, imagine writing a loop that reads a series of characters from a file. Regardless of whether all the reads are successful, the file should be closed. This could be done in a finally block, as shown in Figure 10-14.

```
public class FileCopy {
  public static void main( String args[] ) {
    BufferedInputStream in = null;   // File to copy
    BufferedOutputStream out = null; // File to write to
    int data;                        // Data read from file

    try {
      // Open the input file. After the code is executed, the
      // reference variable in refers to the input file
      // and can be used to read characters from the file.

      // Code omitted

      // Open the output file. After the code is executed, the
      // reference variable out refers to the output file and can
      // be used to write characters to the file.

      // All of this is done in a try block because a
      // FileNotFoundException is thrown if the files
      // cannot be opened.
    }
    catch ( FileNotFoundException e ) {
      System.err.println( "FileCopy: " + e.getMessage() );
      System.exit( 1 );
    }

    // Now copy the files one byte at a time (note both in and
    // out are open at this point in the program)
    try {
      // Copy the files a byte at a time
      while ( ( data = in.read() ) != -1 ) {
        out.write( data );
      }
    }
    catch ( IOException e ) {
      // Something went wrong during the copy
      System.err.println( "FileCopy: " + e.getMessage() );
    }
    finally {
      // Whether the copy succeeded or not, the input and output
      // files must be closed. This is done in the finally
      // block, which is always executed.
```

continued

```
      // The statements that close the files must be placed in a
      // try block because they may throw an exception. In this
      // case the exception is ignored.


      try {
        out.close();
      }
      catch ( IOException e ) {}

      try {
        in.close();
      }
      catch ( IOException e ) {}
   }
  }
} // FileCopy
```

[FIGURE 10-14] Using a finally block to close files

A try block defines scope like any other block in Java. Thus, you can declare variables that are local to the try block. These variables are not known outside the block, and therefore cannot be used outside the try block or any of its associated catch blocks. You must also be careful when initializing local variables inside of a try block, as shown in Figure 10-15.

```
public static void main( String args[] ) {
  int value; // Cannot be declared in try block in order to be visible
             // to the print statement outside of the try block

  try {
    // The method parseInt() throws a NumberFormatException
    // if it is given a string that cannot be interpreted
    // as a number (i.e., "one").
    value = Integer.parseInt( args[ 0 ] );
  }
  catch ( NumberFormatException e ) {
    System.out.println( "Bad argument" );
  }

  System.out.println( value );
}
```

[FIGURE 10-15] Using local variables in a try block

Because the Java compiler enforces the rule that variables must be initialized before they are used, the code in Figure 10-15 does not compile because the compiler cannot guarantee that `value` will be initialized. If the `parseInt()` method throws a `NumberFormatException`, the assignment to `value` is never completed, and the output operation is not meaningful. The correct procedure in this situation is to initialize `value` when it is declared so that it contains a value before the try block is executed, as in the following:

```
int value = 0;
```

The method that throws the exception may not have sufficient information to know the correct action to take. If so, both the `try` statement and the catch block should be placed in the calling method rather than in the method where the exception may be thrown, because the calling method may have a better idea of the proper action to take.

The examples in this chapter have used `try` statements to "account" for the exceptions a program might generate. You can also account for exceptions not by actually catching them, but by declaring that a method may throw an exception. To accomplish this, you can list the exception that can be thrown in a method's **throws clause**, which immediately follows the method's declaration.

Consider the definition of the `pop()` method in Figure 10-16. An attempt to access an element from an empty stack causes a `StackUnderflowException` to be thrown. There is no `try` statement in this method to account for the exception because the `pop()` method has chosen not to handle it. Instead, the programmer placed a `throws StackUnderflowException` clause in the method header that instructed the compiler to look for the `try` statement in the method that called `pop()`.

```
public Object pop() throws StackUnderflowException {
  Object retVal = null;

  if ( tos == null ) {
    throw new StackUnderflowException();
  }

  retVal = tos.data;
  tos = tos.next;

  return retVal;
}
```

[FIGURE 10-16] Declaring exceptions in a method header

SOMETIMES THERE IS NO USER TO ASK WHAT TO DO!

After reading this chapter and starting to understand exceptions and how they work, you are probably asking why you should go to all this trouble when something bad happens. You might think you can just print an error message and ask the program user what to do. This approach probably works for the programs you write as a student, but you will not always write programs that communicate with a user. Consider a pacemaker, an implanted device to help regulate a person's heartbeat. If something goes wrong with this device, it does not have the luxury of asking the user what to do. The pacemaker must be robust enough to detect the error, take corrective action, and continue functioning.

A pacemaker uses an embedded computer system, a system that is completely encapsulated by the device it controls. You have worked with many devices that have embedded systems, including ATMs, cell phones, copiers, MP3 players, and digital video recorders. These systems are dedicated to a specific task and therefore can be optimized to reduce size and cost. Embedded systems are now commonly used in consumer electronic devices.

You might wonder what an embedded system and Java have in common. Java was originally designed to support the development of "smart" consumer electronic devices. Reliability was a great concern because a device that had to be rebooted periodically was not acceptable. This concern was the motivation for several unique features of Java, including exceptions. Exceptions allowed a programmer to direct the system to do more than simply ask the user what to do in case of an error. By developing appropriate exception handlers, an embedded system could often take corrective action and recover automatically from errors without user intervention.

So, the next time you use your cell phone, withdraw money from an ATM, or listen to one of your favorite songs on your MP3 player, remember that exceptions help each of these systems work reliably well.

10.3 Design Guidelines and Examples

10.3.1 Exceptions

This chapter has discussed the syntax and semantics of exceptions and exception handling in Java. Once you understand the basic rules of Java, it is not difficult to write code that can successfully deal with exceptions. You now know that if you invoke a method that throws a checked exception, you must either handle the exception via a `try` statement and a catch block or declare the exception using a throws clause. The basic mechanisms that define, throw, and handle exceptions in Java are not difficult to understand. However, several difficult design questions have yet to be answered. For example, when writing a method, how do you decide when to throw an exception? If you do decide to throw an exception, what type should you throw? Which exception should you throw, and when? What action is appropriate when an exception occurs?

The basic guideline for deciding whether to throw an exception is first to distinguish between special cases or conditions that are part of the algorithm's normal logic and exceptional cases that are best handled using the exception mechanisms of Java. Special cases of the algorithm are usually handled by the program itself so execution can continue. Exceptional cases represent situations in which continued program execution is often not possible, and may be best handled via the Java exception mechanism.

For example, consider a method that reads a single byte from a file each time it is invoked. What should this method do if a hardware error makes it impossible to read the current byte from the file? Clearly, the method cannot fix this problem, and the normal flow of program execution would not be expected to continue. This condition would best be handled by an exception. Now consider what to do if an attempt is made to read past the end of the file. Clearly, no more bytes can be read from the file, but is this really a fatal program error in which normal program flow should be terminated? Probably not, so a reasonable way to handle this case might be to return a special value, such as –1, to indicate that the end of the file has been encountered, but allow the program to continue execution. Generating an exception in this situation is probably not necessary.

Unfortunately, things are rarely this clear cut. Again consider reading bytes from a file, but now assume that the file consists of *N* records, each containing exactly 256 bytes, making the file size an even multiple of 256. Should this method throw an exception when it encounters the end of the file? Based on what we have said so far, the answer seems to be *No*. Instead, we could return –1 to indicate that the end of the file has been reached. However, because this method deals with blocks instead of single bytes, we have to consider where the end-of-file condition occurred. If it occurs in the middle of a block, the size of the file being read was not a multiple of 256 bytes, and therefore was not in the proper format. This is beyond the control of the algorithm and should probably be handled by throwing the

appropriate exception. However, encountering the end-of-file condition before reading a new block or after reading an entire block means the file did have the proper format and should not cause an exception to be thrown.

It can be difficult to decide when to use the Java exception-handling mechanism and when to incorporate the code to handle the situation. Your decision must be based on the program's specifications and requirements, and by determining whether this is part of the algorithm's logical flow or an exceptional condition that is separate from the problem being solved.

After you decide to throw an exception, you must next decide what type of exception to throw. Java provides two general categories of exceptions—unchecked and checked; the only difference between them is whether the compiler forces the programmer to handle the generated exceptions. Remember that one motivation behind exception handling is to provide a means for the compiler to ensure that the programmer properly handles any exceptional cases that arise. This is why the Java language enforces a catch or declare policy. Thus, you should throw a checked exception when the condition that led to the exception is something the program should handle. This ensures that the users of your code do not become lazy and simply assume everything will work. If you throw a checked exception, you or the users of your class must deal with it. Almost all the exceptions you throw are checked exceptions.

The last decision you need to make is which exception to throw. When making this decision, you should avoid forcing the programmer to examine the internal state of the exception to determine exactly what error condition caused the exception to be thrown. The catch blocks that follow a `try` statement are designed to allow the programmer to specify exactly what actions to take based on the exception that was generated. Thus, in most cases, you should design narrower and error-specific exception classes to precisely identify the problem that occurred.

For example, take a second look at the exception classes defined in Figure 10-8. The `StackUnderflowException` class clearly indicates that an underflow condition in the `Stack` class caused the exception to be thrown. We could also have used the more general `StackException` object, which contains the message *Stack underflow*, to indicate what happened. However, this would require the programmer to examine the message within the `StackException` object to determine the cause of the problem. Using multiple exception types that are each designed to identify a specific error gives the programmer much more flexibility. Using catch blocks specific to each exception type, a programmer can choose to handle each kind of exception individually, catch an entire group of related exceptions with a single catch block, or ignore some exceptions altogether.

10.3.2 Handling Exceptions

It is difficult to provide guidance for the best way to handle exceptions in software design, because they often depend on the situation and application. It might be appropriate to handle the exception by terminating the program, executing a modified form of the algorithm,

or ignoring the exception altogether. In this section, we provide different examples for how you might handle exceptions.

Although it seems contrary to the philosophy of exceptions, it is sometimes appropriate to ignore exceptions entirely. However, remember that Java does not allow you to ignore a checked exception; you must "handle" it by specifying an empty catch block that does nothing. Consider the code in Figure 10-17, which prints the sum of its command-line arguments.

```
public class SumArgs {
  public static void main( String args[] ) {
    int total = 0;

    for ( int i = 0; i < args.length; i++ ) {
      try {
        total = total + Integer.parseInt( args[ i ] );
      }
      catch ( NumberFormatException e ) {
        // Ignore non-integer command line args
      }
    }

    System.out.println( "The sum is : " + total ); }
}
```

[FIGURE 10-17] Example of ignoring an exception

In this program, the `parseInt()` method converts the strings from the command line to integer values so they can be added to the total. However, what if the string that `parseInt()` is trying to convert does not contain a properly formatted integer value (e.g., "three"), and a `NumberFormatException` is thrown? We may choose to terminate processing altogether, which in some cases may be the most appropriate response. However, if we encountered a string on the command line that could be converted into an integer, we could ignore the string and continue processing with the next command-line argument, as shown in Figure 10-17. In that program, the output from the call:

```
java SumArgs 1 two 3 4
```

would be 8, the sum of the three valid arguments 1, 3, and 4.

In some situations, the best way to handle an exception may be to throw a different exception, as shown in Figure 10-18.

```
public T pop() throws StackUnderflowException {
  T retVal = null;

  try {
    retVal = tos.data;
    tos = tos.next;
  }
  catch ( NullPointerException e ) {
    throw new StackUnderflowException();
  }

  return retVal;
}
```

[FIGURE 10-18] Example of rethrowing an exception

As discussed earlier, if `Stack` is empty, `tos` is **null**, and any attempt to use `tos` to refer to an object results in a `NullPointerException` being thrown. If the caller of the `pop()` method saw the `NullPointerException`, it would break encapsulation because the caller of the method should not know how the method is actually implemented. It would also be difficult for the caller to determine what caused the exception. The only reasonable assumption would be that the method's implementation has a serious problem, but in fact an attempt was made to remove an element from an empty stack. The proper procedure in this case is to notify the caller of the error by throwing a `StackUnderflowException`, which is what the catch block does if activated.

As you can see, it is difficult to produce hard-and-fast rules about the best way to handle an exception condition. It depends on the specific problem being solved and the event that occurred. You should consider some of the following design options:

1. Write a specific catch block to handle the error condition appropriately for the application.
2. Declare the exception in a throws clause, and let the calling program decide what to do with the condition.
3. Ignore the exception by writing an empty catch block ({}) and let normal processing continue with the code after the try block.
4. Catch the exception and handle it by throwing another exception that is handled by the calling program.

Although this section discussed the fundamental concepts of exceptions, the best way to learn how to use them is to study programs that contain them. In the next section, we introduce the Java classes that allow programmers to create methods that input and output data from external sources. These methods use the exception-handling concepts just presented.

After reading this section, you should have a better understanding of exceptions in Java and how Java programs deal with external devices.

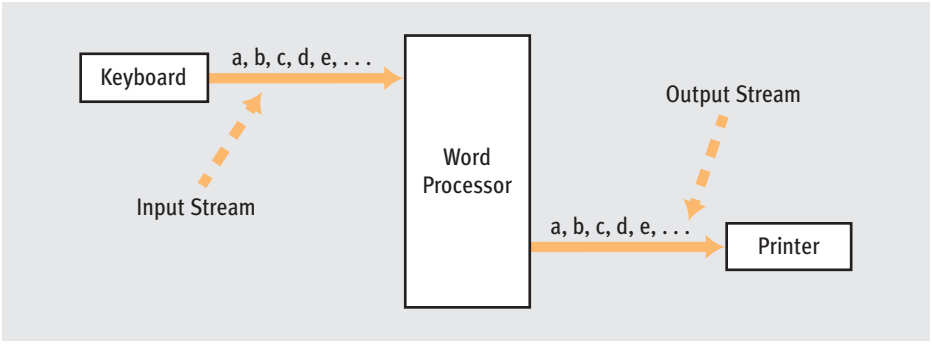
10.4 Streams

10.4.1 Overview

Up to this point, all the Java programs we have discussed in this book have worked with data from internal sources. Most modern software applications, however, process information that resides on external sources, such as a local file, a network connection, a serial connection, or even another program. For example, a word processor reads text as a series of keystrokes from a file and either writes it back to disk for long-term storage or prints it on standard output.

Although a keyboard, disk, and printer are dramatically different types of devices, a program deals with them in the same way. When reading from a disk, for example, the program inputs one piece of data at a time from the device in a serial fashion. When reading from a keyboard, the characters typed by the user are sent to the program one character at a time and are processed in the order they were typed. You can think of input as a stream of data that flows from the keyboard to the program, in the same way that water flows in a stream.

Many modern object-oriented languages use a programming abstraction called a **stream** to deal with external devices in a generalized way. A stream carries an ordered sequence of data of undetermined length. You can think of a stream as a pipe that connects a source of data to a user of the data. The data flows through the stream, just like water flows through the pipes in your house. The stream model is commonly used in class libraries that provide access to external devices such as disks, keyboards, or printers. Figure 10-19 illustrates two of the streams that might be associated with a word-processing program.



[FIGURE 10-19] Examples of streams

A stream connects an entity that generates data, referred to as the **source**, to an entity that reads the data, called a **sink**. The data in the stream flows in one direction, from the source to the sink. The source places data into the stream one element at a time, and the sink reads the data one element at a time in the exact order they were generated by the source. A program can serve as either a source or a sink of data. The difference lies in whether the program is generating information to be placed into a stream or is reading information from a stream. A single program commonly works with more than one stream, and it may act as a source for some streams and a sink for others.

Before a program can use a stream, you must establish the identity of the entities that reside at the two endpoints, a process known as **opening a stream**. For example, before you can use the input stream in Figure 10-19, one end of the stream must be associated with the keyboard and the other must be associated with the word-processing program. In most libraries, one end of the stream is associated with the program that created the stream. The program then specifies the device that is associated with the other end of the stream, either when the stream is created or opened. For example, in Figure 10-19, the word-processing application may have created the input stream and specified that the keyboard is the other end of the stream. Once a stream has been opened, data can flow from the source to the sink. Data transfer continues from the source to the sink until no more data is left to transfer.

Because the length of the data carried on the stream is not specified, the source must have a way of signaling the sink when there is no more data to send. The source **closes** the stream to indicate that it has no more data. In a typical application, the sink enters a loop and continues to read characters from the stream until it detects that the stream has been closed. It then exits the loop and closes the stream on its end. The general algorithms to read information from a stream or write information to it are shown in Figure 10-20.

Sink

```
Open the stream
While there is data to write {
    Prepare the next piece of data
    Write the data to the stream
}
Close the stream
```

Source

```
Open the stream
While there is data on the stream to read {
    Read next piece of data from the stream
    Process the data
}
Close the stream
```

[FIGURE 10-20] Algorithms for using streams

When a stream is closed, the operating system that manages the transfer of information between devices is informed that the stream is no longer required. This allows it to release any resources it allocated to implement the stream. A common mistake of many programs is forgetting to close streams when they are no longer needed. Failing to close a stream can result in an incomplete transfer of information across the stream or, worse yet, can allocate operating system resources to streams that are no longer in use.

The Java API defines classes that provide many types of streams programmers can use to deal with external devices. The classes that implement streams are contained in the `java.io` package. Each stream class provides methods that allow you to open, read from, write to, and close a stream, making it possible to write programs that implement the algorithmic style of Figure 10-20. The next section describes some of the more common stream classes in the `java.io` package.

10.4.2 The `java.io` Package

The `java.io` package contains classes that provide several ways to access stream-oriented data. The stream classes in the package can be divided into two categories based on whether they deal with character-based or byte-based data. Although the data being read by either type of stream always consists of byte data, the character-based streams convert these bytes into characters using the default character encoding scheme on the platform the program is using. Table 10-3 lists some of the important character- and byte-oriented streams in the `java.io` package.

CHARACTER STREAMS	BYTE STREAMS	DESCRIPTION
`Reader` `Writer`	`InputStream` `OutputStream`	Base classes for all streams
`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`	Implements buffering to provide efficient access
`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`	Streams that can read to or write from memory
	`DataInputStream` `DataOutputStream`	Allows a program to read or write primitive Java types directly
`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`	Streams to read from files or write to files
`InputStreamReader` `InputStreamWriter`		Converts byte-based data to character-based data
`LineNumberReader`		A buffered stream that keeps track of line numbers
`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`	Channels the output from one thread into the input of another

continued

CHARACTER STREAMS	BYTE STREAMS	DESCRIPTION
`PrintWriter`	`PrintStream`	Allows you to print different types of primitive data
`StringReader` `StringWriter`		Reads from or writes to standard Java `String` objects

[TABLE 10-3] `java.io` classes

All of the stream classes in the `java.io` package are subclasses of one of the following four abstract classes: `Reader`, `Writer`, `InputStream`, or `OutputStream`. Classes that inherit from `Reader` or `Writer` are character-based streams, and the classes that extend `InputStream` or `OutputStream` are byte-based streams. You should note from Table 10-3 that the names of all the character-based stream classes end in either `Reader` or `Writer` and that the names of the byte-oriented streams, with the exception of `PrintStream`, end with either `InputStream` or `OutputStream`. This naming convention makes it easy to determine the type of data with which a particular stream class works.

Through the proper use of inheritance, you can write a program that does not depend on the actual type of stream it uses. For example, if you wrote a method that took as a parameter a reference to a `Reader`, you could pass it any subclass of `Reader`, and it would still function correctly. Thus, the same method could be used to read information from a file, network connection, or serial line without having to be modified in any way, because each of the subclasses overrides the methods in the superclass that actually read the data. This makes it possible for the class to work correctly on the specific device for which it was designed.

The stream classes can be further subdivided based on whether they serve as source/sink streams or processing streams. **Source/sink streams** are connected directly to an external source, and read or write directly to or from that source. An example of a source/sink stream is the `FileReader` class, which can read character-based data from a file. Source/sink streams are used in Java to read data from a variety of different devices, including files, keyboards, printers, and network connections. Some of the more common source/sink streams—file, memory, and pipe—are listed in Table 10-4.

In Table 10-4, the file type identifies streams that can read data from and write data to a file. The file streams are probably the ones you use most often in your Java programs; we discuss them at length later in this chapter. Memory streams read and write data to and from memory using stream-based semantics. This type of stream is useful when dealing with certain types of network data, and is discussed further in Chapter 13. Finally, pipe streams allow the output of one thread to be the input of another thread.

SOURCE/SINK TYPE	CHARACTER STREAMS	BYTE STREAMS
File	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`

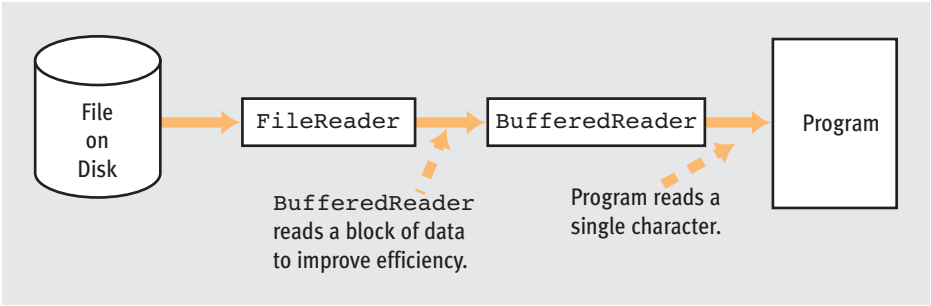
Memory	`CharArrayReader`
	`CharArrayWriter`
	`StringReader`
	`StringWriter`
		`ByteArrayInputStream`
		`ByteArrayOutputStream`

Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

[TABLE 10-4] Java source/sink streams

A **processing stream** performs some transformation operation on data as it flows through the stream. For example, in Figure 10-21, a `BufferedReader` has been placed between a program and a `FileReader` stream that is used to read character data from a disk file. The purpose of the `BufferedReader` is to buffer the read requests made by the program so that the information can be read from the disk as efficiently as possible.

Streams are often linked together, as shown in Figure 10-21, to perform a variety of operations on the data as they flow through the stream. This process of linking streams together in a pipeline fashion is often called **wrapping streams** around each other. In Figure 10-21, a `BufferedReader` has been wrapped around a `FileReader` to provide buffering in the data pipeline. Wrapping streams allows you to combine the beneficial features of several different streams.



[FIGURE 10-21] Using a processing stream

Table 10-5 lists some of the more common processing stream classes. Given this description of the stream classes in the `java.io` package, you should be able to determine the correct class for a given situation. You first need to determine the type of data your program should use and then the type of the source or sink you want to use. With this information, it is relatively easy to

determine the name of the class that provides the functionality you require. For example, a program that needs to read character information from a file would use a `FileReader`, and a program that writes byte data to memory would use a `ByteArrayOutputStream`.

Identifying the name of the class you use to implement a stream is only the first step in writing your program. After selecting the appropriate class, you must write the code to actually create the stream and read or write the information. In the next section, we explain how to use the Java stream classes by presenting several sample programs that read or write information using the classes.

PROCESS	CHARACTER STREAM	BYTE STREAM
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Byte/character conversion	`InputStreamReader` `InputStreamWriter`	
Data conversion		`DataInputStream` `DataOutputStream`
Printing	`PrintWriter`	`PrintStream`
Counting	`LineNumberReader`	

[TABLE 10-5] Processing streams

10.4.3 Using Streams

This section presents programs that use the Java stream classes to read data from a file, read information from the keyboard, and write data to a file. The first program that we discuss, `FileEcho`, takes the name of a text file from the command line and prints the contents of the file to standard output. Because the program is working with a text file made up of characters, it uses a `FileReader` object to read the contents of the file.

All of the `Reader` and `InputStream` classes provide a `read()` method that can read the next piece of data from the stream. You might be surprised that the signatures for the `read()` method are the same in both classes, even though a `Reader` works with character-based streams and an `InputStream` works with byte-based streams. The `read()` method returns an integer value that contains the data read from the stream. The method does not return until either the data is available, the end of the stream is detected, or an exception is thrown.

The results returned by the `read()` method are interpreted differently depending on whether you are using an `InputStream` or a `Reader`. An `InputStream` always reads the next byte (8 bits) of data from the stream. This 8-bit value is returned as an integer and is

in the range from 0 to 255. The number of bytes read by a `Reader` depends on the character encoding used by the platform on which the program runs. If the characters are encoded using 8-bit ASCII, one byte is read. If the characters are encoded using 16-bit Unicode, two bytes are read. The `Reader` automatically determines which character code to use when it is created. The method `read()`, when invoked on a `Reader`, returns the character code of the next character read from the stream as a 16-bit integer value in the range from 0 to 65,535.

The program `FileEcho`, shown in Figure 10-22, illustrates how to use a `FileReader` to read characters from a file. Recall from Figure 10-20 that a stream must be opened before any data can be read. The process of instantiating a new `FileReader` object opens the stream and associates it with a file. The `FileReader` class provides three constructors. The one-parameter form in Figure 10-22 takes a string, which is the name of the file to open. If the file cannot be opened because it does not exist or you do not have permission to read it, the constructor throws a `FileNotFoundException`.

Once the file is opened, the program uses the `read()` method to obtain characters, one at a time, from the file. Because the `read()` method returns an integer value instead of a character, the program must read the character into an integer variable and then cast the value to a character before printing it. If you fail to do this, the program prints the numerical value of the character code instead of the characters it represents—in other words, instead of printing an *a*, the program outputs the integer 97.

The loop that reads characters from the file executes until `read()` returns –1. When `read()` is invoked on a stream that has been closed by the source (in other words, there is no more data), it returns –1. Because –1 is outside the range of valid values returned by `read()` (0–255 for an `InputStream` and 0–65535 for a `Reader`), you do not confuse the closed-stream value with a valid data value.

```
import java.io.*;

public class FileEcho {
  public static void main( String args[] ) {
    FileReader in = null;
    int ch;

    // Make sure the number of arguments is correct
    if ( args.length != 1 ) {
      System.err.println( "Usage:  FileEcho sourceFile" );
    }
    else {
      try {
        // Attempt to open the file for reading
        in = new FileReader( args[ 0 ] );
```

continued

```
        // While characters are left in the file to read,
        // read one and echo it to the screen. Note the cast on
        // the invocation of the method read(). Without it, this
        // program will print bytes (i.e., numbers).
        // Remember bytes != characters
        while ( ( ch = in.read() ) != -1 ) {
          System.out.print( (char)ch );
        }
      }
      catch ( IOException e ) {
        // An I/O error occurred or the file could not be opened
        System.err.println( "FileEcho:  " + e.getMessage() );
      }
      finally {
        // Close the file in a finally block. This way, whether
        // an I/O exception is thrown or not, the file is
        // closed. The file must be closed in a try block
        // because close() throws an IOException.
        try {
          if ( in != null ) {
            in.close();
          }
        }
        catch( IOException e ) {
          System.err.println( "FileEcho:  " + e.getMessage() );
        }
      }
    }
  }
} // FileEcho
```

[FIGURE 10-22] `FileEcho.java`

The stream used by the program to read the file is closed in the finally block of the try block. Recall that the finally block is always guaranteed to execute. By writing the program in this way, you can be confident that the stream is always properly closed, whether the program succeeds or an exception occurs during execution. The check to determine if the stream variable `in` is **null** is required because when the finally block is activated, we do not know exactly what has happened in the program. If an exception occurred during invocation of the constructor, `in` is **null**. If the file was successfully opened and the program either terminates successfully or an exception occurs while reading the file, the variable `in` refers to the stream object being used to access the file. If we attempt to invoke the `close()` method using a **null** reference, a `NullPointerException` is thrown.

Note how the exceptions were handled in Figure 10-22. There is only one try block, and the only catch block associated with the block handles an `IOException`. Recall that the

constructor, which opens the file and associates it with a stream, may throw a `FileNotFoundException`. Because this exception is a subclass of `IOException`, the single catch block for `IOException` matches either exception. The only consequence of writing the code in this manner is that you cannot determine which type of exception occurred, based only on the catch block that is activated. However, in this program, it does not matter; the only action if an exception occurs is to print an error message and terminate the program. If each exception type had to be handled differently, then two separate catch blocks would have been used.

Reading data from a file character by character is inefficient. When working with a disk file, it is much more efficient to read or write an entire block of information at a time—that is, one complete disk sector. Essentially, it takes the same amount of time to read or write an entire block of data from a disk file as it does to read or write a single character. Therefore, most programs that work with data stored on a disk will buffer the reads and writes. This feature allows you to save up, or **buffer**, several read requests and then execute a single read operation to obtain all of the data from the disk. Thus, although the program in Figure 10-22 works, it is not very efficient.

Changing the program in Figure 10-22 so that it uses buffering is trivial in Java. Recall that one of the processing streams provided in the `java.io` package is the `BufferedReader` class. A `BufferedReader` can be placed between a `FileReader` and a program to automatically buffer the read requests so that the program executes more efficiently. The program in Figure 10-23, `BufferedFileEcho`, uses a `BufferedReader` to read data from the file. The streams in this program are connected, as shown in Figure 10-21.

```
import java.io.*;

public class BufferedFileEcho {
  public static void main( String args[] ) {
    BufferedReader in = null;
    String line = null;

    // Make sure the number of arguments is correct
    if ( args.length != 1 ) {
      System.err.println( "Usage:  BufferedFileEcho sourceFile" );
    }
    else {
      // Attempt to open the file for reading. Note that this
      // program does some additional "plumbing". It creates
      // the FileReader and then wraps a BufferedReader around
      // it. The program reads from the BufferedReader, which
      // in turn reads from the FileReader.
```

continued

```
      try {
        in = new BufferedReader( new FileReader( args[ 0 ] ) );

        // While lines are left in the file, read a line and
        // print it on the screen. Note that readLine() strips the
        // line termination character(s) from the input, which is why
        // the lines are printed using println().
        while ( ( line = in.readLine() ) != null ) {
          System.out.println( line );
        }
      }
      catch ( IOException e ) {
        // Either an I/O error occurred or the file could not be opened
        System.err.println( "BufferedFileEcho:  " + e.getMessage() );
      }
      finally {
        // Close the file in a finally block. This way, whether
        // an I/O exception is thrown or not, the file is closed.
        // The file must be closed in a try block because
        // close() throws an IOException.
        try {
          if ( in != null ) {
            in.close();
          }
        }
        catch( IOException e ) {
          System.err.println( "BufferedFileEcho:  " + e.getMessage() );
        }
      }
    }
  }
} // BufferedFileEcho
```

[FIGURE 10-23] `BufferedFileEcho.java`

Only two major changes have been made in the program of Figure 10-23. The first change involves the statement that invokes the constructor that creates the streams. In Figure 10-23, the streams are created by the following statement:

```
in = new BufferedReader( new FileReader( args[ 0 ] ) );
```

This is an example of the stream wrapping concept mentioned in the previous section. Notice how the constructor for the `BufferedReader` is wrapped around the constructor for `FileReader`. This declaration creates two streams: a `FileReader` and a `BufferedReader`. The `FileReader` is created as before. Its constructor takes as its only parameter a string

containing the name of the file to read. However, the reference to the `FileReader` object is not assigned to the stream variable `in` but is passed as a parameter to the constructor for `BufferedReader`. This causes the `BufferedReader` object to read information from the `FileReader`. When the program reads from the stream referred to by `in`, the request is passed to `BufferedReader`, which immediately returns the information if it is in the buffer. Otherwise, it reads the next block of data from `FileReader` and passes the next character on to the program.

The second change to the program in Figure 10-23 involves the way data is read from the file. In the `FileEcho` program, the data was read one character at a time using the `read()` method. In this program, the data is read using `readLine()`, a method that reads an entire line of characters from the file. A text file can be viewed as consisting of lines of characters that are separated by a line termination character, and the actual character that marks the end of a line may vary from platform to platform. The `BufferedReader` class automatically determines and uses the correct line termination characters for the platform on which it is running. However, these line termination characters are not read by `readLine()` because they only function to mark the end of a line. Therefore, you must print the lines in the file using the `println()` method instead of the `print()` method. Finally, the `readLine()` method returns a **null** reference when it reaches the end of the stream.

Note that we could have written the program in Figure 10-23 to use the `read()` method instead of the `readLine()` method. In either case, the `BufferedReader` would still perform the buffering necessary to improve the program's efficiency. The `readLine()` method makes it slightly easier to print the file and reduces the number of times the while loop has to execute.

Finally, note in Figure 10-23 how the streams are closed. The `close()` method in the finally block is invoked only on the `BufferedReader`, not on both streams. The `BufferedReader` will invoke `close()` on the stream to which it is connected, namely `FileReader`. In general, when you have wrapped streams around each other to read data, you only need to close the "outermost" stream—`BufferedReader` in this example. This stream, in turn, invokes `close()` on the streams around which it is wrapped.

One consequence of working with a buffered stream is that you don't always know exactly what has been written to the sink at the end of the stream. For example, you may execute code that writes a few characters to a file, but unless you filled the buffer, the data is not actually written to the disk. Instead, it is held in the buffer until the buffer fills, at which point the data is written to the disk. You think the file contains the characters that your program wrote to the file, but the characters are actually still in the computer's memory waiting to be written to disk. Normally, you do not need to worry about a lack of synchronization between your program and the disk file you are writing to, but such synchronization is sometimes important.

When a buffered stream is closed, any information stored in a buffer is automatically written to the sink. This is another reason you must close streams when you are finished with

them. The buffered streams in the `java.io` package provide a `flush()` method that forces a reader to write the contents of its buffer, regardless of the number of bytes being held in it. You should invoke `flush()` in your program when the contents of any buffers in the stream must be written to the sink so that the program's view of the sink and the actual data written to the sink agree.

Our third example shows how to make a copy of a file stored on disk. The `FileCopy` program of Figure 10-24 takes the name of two files entered on the command line. It copies the entire contents of the first file named to the second file. If the second file already exists, its contents are overwritten.

```
import java.io.*;

public class FileCopy {
  public static void main( String args[] ) {
    BufferedInputStream in = null;
    BufferedOutputStream out = null;
    int data;

    // Check command line arguments
    if ( args.length != 2 ) {
      System.out.println( "Usage:  FileCopy sourceFile destFile" );
    }
    else {
      try {
        // Open the input file. A BufferedReader is used here to
        // make the copy more efficient.  The copy could have
        // been done using the FileInputStream only.

        // The FileInputStream was used instead of a FileReader
        // because the program is interested in copying bytes
        // and not necessarily characters/strings.
        in = new BufferedInputStream(
               new FileInputStream( args[ 0 ] ) );

        // Take care of the output side.  If the output
        // file exists, it is overwritten.
        out = new BufferedOutputStream(
                new FileOutputStream( args[ 1 ] ) );

        // Copy the files a byte at a time
        while ( ( data = in.read() ) != -1 ) {
          out.write( data );
        }
      }
```

continued

```
      catch ( FileNotFoundException e ) {
        System.err.println( "FileCopy:  " + e.getMessage() );
      }
      catch ( IOException e ) {
        System.err.println( "FileCopy:  " + e.getMessage() );
      }
      finally {
        // The files are closed in the finally block so that no
        // matter what happens, the files are closed. Two try
        // blocks are used in case the first close fails.
        try {
          if ( in != null ) {
            in.close();
          }
        }
        catch ( IOException e ) {
          System.err.println( "FileCopy:  " + e.getMessage() );
        }

        try {
          if ( out != null ) {
            out.close();
          }
        }
        catch ( IOException e ) {
          System.err.println( "FileCopy:  " + e.getMessage() );
        }
      }
    }
  }
} // FileCopy
```

[FIGURE 10-24] `FileCopy.java`

The `FileCopy` program in Figure 10-24 is quite similar to the other programs discussed so far. The most significant difference is that the program writes the contents of the first file to another file instead of to standard output. This significantly changes the behavior of the program, but its only effect on coding is that instead of writing to the screen, it writes to a `FileWriter` output stream. The basic structure of the program does not change. The program uses an `InputStream` instead of a `FileReader` so that it can copy either text files or data files. The program uses buffered streams so that it reads and writes the files as efficiently as possible.

The finally block in Figure 10-24 closes the two streams—the stream that reads the file and the stream that writes the file—in two separate try blocks, because an exception may be thrown when we attempt to close a stream. If both `close()` statements were in the same try block and the first `close()` statement threw an exception, then the second `close()` statement would never be executed, leaving the second file open and wasting system resources.

The examples discussed thus far have not processed the data in the file in any meaningful way, except to print them or copy them to another file. The program in Figure 10-25, `ReadNums`, illustrates how easily you can modify a program that reads from a file so that it does some type of transformation on the data it reads. In this program, the file being read is assumed to consist of lines containing a single integer value. The program reads the file one line at a time, converts the string representation of the number to an integer, and computes the sum of the numbers. When all the numbers in the file have been read, the program prints the sum, closes the file, and terminates.

```
import java.io.*;

public class ReadNums {
  public static void main( String args[] ) {
    BufferedReader in = null;
    String line = null;
    int sum = 0;

    // Make sure the number of arguments is correct
    if ( args.length != 1 ) {
      System.err.println( "Usage:  ReadNums sourceFile" );
    }
    else {
      try {
        // Attempt to open the file for reading
        in = new BufferedReader( new FileReader( args[ 0 ] ) );

        // Read the numbers a line at a time from the source file
        while ( ( line = in.readLine() ) != null ) {
          // Attempt to convert the string to an int. If it
          // can't be done, ignore the line
          try {
            sum = sum + Integer.parseInt( line );
          }
          catch ( NumberFormatException e ) {}
        }

        // Numbers are read and summed; print the results
        System.out.println( "The sum is " + sum );
      }
      catch ( IOException e ) {
        System.err.println( "ReadNums:  " + e.getMessage() );
      }
```

continued

```
      finally {
        try {
          if ( in != null ) {
            in.close();
          }
        }
        catch ( IOException e ) {
          System.err.println( "ReadNums:  " + e.getMessage() );
        }
      }
    }
  }
} // ReadNums
```

[FIGURE 10-25] `ReadNums.java`

At this point, you should realize that when you execute `System.out.print()` or `System.out.println()` to print from your program, you are using a stream. If you examine the `System` class defined in the `java.lang` package, you see that the class defines three class variables: `out`, `in`, and `err`. These variables refer to streams that can print information to the screen, read information from the keyboard, or print information to the error stream.

The class variables `out` and `err` refer to two different `PrintStreams` that are connected to the standard output device and the standard error device, respectively. **Standard output** refers to the device that normally displays information, which for most computing systems is the screen. The **standard error** stream prints diagnostic messages. The idea behind having two streams is to separate the "normal" output from a program from the diagnostic warning and error messages that may be printed while the program is executing. However, in most cases, the standard output and standard error devices are one and the same. All the error messages in the catch blocks in this section were printed using the standard error stream.

You should now understand that the `System.out` in `System.out.println()` refers to the `PrintStream` object connected to the standard output device, and that you are invoking the `println()` method on that object. If you examine the Javadoc pages for the `PrintStream` class, you see that the class provides many versions of the `print()` and `println()` methods. The primary difference between these methods is the type of data they print. For example, invoking `System.out.println(10)` in a program executes the version of `println()` that takes an integer argument. This version of the `println()` method converts the value 10 to the appropriate sequence of characters and sends the result to the screen.

The class variable `in` refers to an `InputStream` object that is connected to the standard input device. On most computers, **standard input** refers to the keyboard. However, it does not matter what the standard input stream is connected to because your program still reads it as a stream, regardless of the device type. You might be surprised to find that `in` refers to an `InputStream` as opposed to a `Reader` because we typically think of the keyboard as a

character-oriented device. As a consequence of this decision by the designers of the Java class libraries, you cannot read character-based data from the keyboard unless you first convert the byte data coming in from `System.in` to character data. If you wrap an `InputStreamReader` around `System.in`, you can read the keyboard data as characters.

The `InputNums` program of Figure 10-26 illustrates how to read character data from the keyboard. This program is very similar to the `ReadNums` program in Figure 10-25. The only difference is the stream type used to obtain the input. In this example, `in` refers to a `BufferedReader` that has been wrapped around an `InputStreamReader`, which in turn has been wrapped around `System.in`.

```
import java.io.*;

public class InputNums {
  public static void main( String args[] ) {
    BufferedReader in = null;
    String line;
    int sum = 0;

    // Attempt to open the file for reading
    try {
      // Connect System.in to a BufferedReader
      in = new BufferedReader( new InputStreamReader ( System.in ) );

      while ( ( line = in.readLine() ) != null ) {
        // Attempt to convert the string to an int.
        // If it can't be done, ignore the line.
        try {
          sum = sum + Integer.parseInt( line );
        }
        catch ( NumberFormatException e ) {}
      }

      // Numbers are read and summed; print the results
      System.out.println( "The sum is " + sum );
    }
    catch ( IOException e ) {
      System.err.println( "InputNums:  " + e.getMessage() );
      System.exit(1);
    }
    finally {
```

continued

```
      try {
        if ( in != null ) {
          in.close();
        }
      }
      catch ( IOException e ) {
        System.err.println( "InputNums:  " + e.getMessage() );
      }
    }
  }
} // InputNums
```

[FIGURE 10-26] `InputNums.java`

Figures 10-25 and 10-26 are identical except that the input is coming from different streams. In Figure 10-25, `in` refers to a `BufferedReader` object that is wrapped around a `FileReader`. This means the program's input is coming from a character-based file. In Figure 10-26, the stream variable `in` again refers to a `BufferedReader` object. The difference is that in Figure 10-26, the `BufferedReader` object ultimately takes its input from `System.in`. In both programs, `in` refers to the same type of stream, a `BufferedReader`; thus, you can combine these two programs into one.

The program in Figure 10-27 shows a method called `sumInput()`, which takes as a parameter a reference to a `BufferedReader` that contains the lines to be summed. The method does not care what type of stream is connected to `BufferedReader`. It may involve files, keyboards, or a network. Regardless of the actual type of device in use, polymorphism makes it possible for `sumInput()` to read the lines on the device and compute the sum.

The main method in Figure 10-27 uses the command line to determine the source of the program input. If an argument is found on the command line, the program attempts to create a `BufferedReader` that is wrapped around a `FileReader` connected to the file specified on the command line. If the command line has no arguments, the program wraps a `BufferedReader` around an `InputStreamReader` that is wrapped around `System.in`. Regardless of how it is actually created, the resulting `BufferedReader` object is passed to the `sumInput()` method to calculate the sum. Due to the polymorphic behavior of the stream classes, the appropriate code is executed when reading the information from the streams.

```
import java.io.*;

public class SumNums {
  public static void main( String args[] ) {
    BufferedReader in = null;
```

continued

```
  // Hook up to System.in or a FileReader depending on
  // the command line argument
  try {
    if ( args.length > 0 ) {
      // Try to hook up to the first thing on the command line
      in = new BufferedReader( new FileReader( args[ 0 ] ) );
    }
    else {
      // Hook up to System.in
      in = new BufferedReader( new InputStreamReader( System.in ) );
    }

    // Sum the input and print the result
    System.out.println( "The sum is " + sumInput( in ) );
  }
  catch ( IOException e ) {
    System.err.println( "SumNums:  " + e.getMessage() );
  }
  finally {
    try {
      if ( in != null ) {
        in.close();
      }
    }
    catch ( IOException e ) {
      System.err.println( "SumNums:  " + e.getMessage() );
    }
  }
}

/**
 * Read lines from the buffered reader and convert the integer values
 * on those lines to numbers. The sum of the numbers is returned.
 * Note that this method takes a buffered reader as a parameter and
 * does not care if the buffered reader is connected to a file or the
 * standard input stream.
 *
 * @param in the buffered reader to process.
 *
 * @return the sum of the numbers in the stream.
 */
public static int sumInput( BufferedReader in ) throws IOException {
  String line = null;
  int sum = 0;
```

continued

```
    // Read the BufferedReader one line at a time
    while ( ( line = in.readLine() ) != null ) {
      // Attempt to convert the string to an int.
      // If it can't be done, ignore the line.
      try {
        sum = sum + Integer.parseInt( line );
      }
      catch ( NumberFormatException e ) {}
    }

    // Numbers are read and summed; return the results
    return sum;
  }

} // SumNums
```

[FIGURE 10-27] SumNums.java

REACHING THE 100-GIGABASE MILESTONE

For many years, biologists have known that DNA provides the information a cell requires to function. Since the 1800s, when Gregor Mendel started studying heredity, scientists have performed thousands of experiments to understand how living organisms function. Thomas Watson and James Crick discovered that DNA is the source of genetic information in living organisms, and that it consists of a sequence of four amino acids, commonly abbreviated as A, T, G, and C. In 1978, Fred Sanger ushered in a new era in biology by developing a technique whereby he could determine the sequence of amino acids in a strand of DNA. Today, biologists routinely sequence entire genomes consisting of billions of nucleotides.

Bioinformatics is a field in which biologists, chemists, computer scientists, and statisticians work to collect and analyze data generated by experimental work in the laboratory. This collaboration has led to the development of several large databases of genetic data. GenBank, maintained by the National Center for Biotechnology Information (NCBI), is one such database. From its beginning, GenBank was designed to be a public repository of sequence information collected from laboratories around the world.

continued

For almost 20 years, scientists have been submitting sequence information to GenBank. By 2005, it contained more than 100 gigabases of sequence data. The 100,000,000,000 bases represent genetic information from more than 165,000 organisms. To put this number into perspective, 100 billion bases is about equal to the number of stars in our galaxy. GenBank doubles in size about every 10 months. Researchers around the world use the information in this database to understand how living organisms function and to understand diseases that affect living organisms.

Although it is amazing to think that a database can hold so much data, even more remarkable is the speed at which you can search it. From a Web interface you can submit a query to the database and receive an answer in just a few seconds. This performance is enabled by sophisticated programming and hardware, but the core of the system uses some of the basic techniques discussed in this chapter. So, while some of the programs in this chapter have been very simple, you could use the same techniques to build the next generation of large database systems.

10.4.4 The `Scanner` Class

All the programs in this chapter have read characters, bytes, or lines from a source stream. Once the data was read from the stream, we had to write the code to convert the data into the appropriate format. For example, consider the `InputNums` program of Figure 10-26. In this program, we read input one line at a time and then used the `parseInt()` method from the `Integer` class to convert the characters into numeric form. Given that data is commonly read from external sources in character or byte form and then converted to a different format in a program, many programming environments provide tools to simplify the process.

The `Scanner` class in the `java.util` package provides several methods that help transform data read from a stream or a string into different formats. A `Scanner` splits its input into a series of substrings, or tokens, which can then be read either as strings or as the primitive types they represent. By default, a `Scanner` defines a token as a sequence of characters delimited by one or more whitespace characters. For example, the following string:

"Grumpy Cat was born in 1996"

consists of the following tokens:

"Grumpy", "Cat", "was", "born", "in", "1996"

It does not matter that more than one space separates "was" and "born" in the original string. The `TextTokenizer` program of Figure 10-28 illustrates how to use a `Scanner` to tokenize the contents of a text file.

```
import java.io.*;
import java.util.*;

public class TextTokenizer {
    public static void main( String args[] ) {
      Scanner in = null;  // Scanner to read input

      // Usage check
      if ( args.length != 1 ) {
          System.err.println( "Usage:  java TextTokenizer file" );
          System.exit( 1 );
      }

      // Attempt to create a scanner attached to a buffered file
      try {
          in = new Scanner(
               new BufferedReader(
                 new FileReader( args[ 0 ] ) ) );
      }
      catch ( FileNotFoundException e ) {
          System.err.println( "TextTokenizer:  error opening file" );
          System.exit( 1 );
      }

      // Read the tokens in the file
      while ( in.hasNext() ) {
          System.out.println( in.next() );
      }

      // Close the scanner
      in.close();
    }
}
```

[FIGURE 10-28] `TextTokenizer.java`

The `TextTokenizer` program begins much like the other programs you have seen in this chapter. The major difference is that instead of reading from the buffered stream directly, a `Scanner` is connected to the stream. Now, you can use the `hasNext()` method to determine if tokens are available for reading and the `next()` method to read the next tokens. This should remind you of the discussion of iterators and the Java collection classes in Chapter 9. The program does not have to worry about finding the spaces that separate the words, or breaking the input into tokens; it simply reads the tokens from the `Scanner`.

For each of the primitive types, the `Scanner` class provides the `hasNext()` and `next()` methods; when used together, they read and return a value of that type from the stream. For example, the `hasNextInt()` method returns true if we have not reached the end of the

stream and determined that the next token to read from the stream can be interpreted as an integer value. The `hasNextInt()` method returns false when you have reached the end of the stream. The `hasNext` methods do not advance the input, so you can call a number of different `hasNext` methods to determine the type of the next token and then read it using the appropriate `next` method (see Figure 10-29).

```
import java.util.*;

/**
 * Demonstrate the hasNext and next methods of the scanner class.
 */
public class ReadInts {
    public static void main( String args[] ) {
      Scanner in = new Scanner( System.in );

      // Read until nothing is left
      while ( in.hasNext() ) {
          if ( in.hasNextInt() ) {
            System.out.println( "int: " + in.nextInt() );
          }
          else if ( in.hasNextDouble() ) {
            System.out.println( "double: " + in.nextDouble() );
          }
          else if ( in.hasNextBoolean() ) {
              System.out.println( "boolean: " + in.nextBoolean() );
          }
          else {
              System.out.println( "unknown: " + in.next() );
          }
      }

        in.close();
    }

} // ReadInts
```

[FIGURE 10-29] Using the `Scanner` class

If you attempt to read a token that does not match the type of the next token in the stream, the `next` method throws an `InputMismatchException`. This could happen, for example, if you invoked `nextInteger()` on a scanner and the next token in the stream was a string. If this type of error occurs, the scanner does not remove the token that caused the mismatch from the stream. In this way, you can determine which `next` method should have been used, and use it to retrieve the token from the scanner.

10.5 Summary

Software systems, like human beings, need to deal with rare and unexpected events. An unexpected event that lies outside the normal behavior of a software algorithm is called an **exception**. Exception handling can help you develop robust and fault-tolerant software. It does not allow errors to be ignored or overlooked, but forces programmers to recognize and deal with exceptional circumstances in the appropriate manner.

In Java, exceptions are objects that inherit from the class `Throwable`. The `throw` statement activates the exception-handling mechanism. Java divides exceptions into two broad categories: unchecked and checked. Any checked exception that is thrown during the execution of a program must either be handled via a `try` statement or declared in a method header using the throws clause. The actual code for handling the exception is contained in the catch block, which is specific to each exception type.

Streams are a common abstraction used in many object-oriented programs to describe how programs interact with external devices. Once a stream is opened, it is connected to a source that generates data and a sink that consumes the data. The sink typically enters a while loop that reads information until the source has closed its end of the stream. At this point, the loop terminates and the sink closes its end of the stream.

The stream classes in `java.io` are categorized by (a) the type of data they work with, either character or byte, and (b) whether the class is a source/sink stream or processing stream. Source/sink streams are used to read or write information to external devices, and processing streams are typically wrapped around other streams to process the data as they flow down the stream. Streams allow programmers to write software that can accept and process data independent of where it comes from—a file, an I/O device, or a network.

BILL GATES

The name Bill Gates is synonymous with the personal computer, and stirs a variety of emotions. Some people consider him a computer pioneer, while others view him as a tyrant or evil genius who does whatever is necessary to build his corporate empire. Regardless of how you feel about him, you have to agree that he is responsible for shaping the face of computing as we know it today.

continued

Gates was born on October 28, 1955, in Seattle, Washington. His family was known for its involvement in business, politics, and community service. Gates attended Lakeside School, Seattle's most exclusive preparatory school, where he was first introduced to computers. He and his friend Paul Allen spent the bulk of their time at Lakeside learning all they could about computers. Determined to find a real-world use for their computing skills, Gates and Allen formed the Lakeside Programmers Group. One of their first jobs was to discover bugs and expose security weaknesses in a commercial computing system.

Gates' life changed after the development of the Altair 8080 by MITS (Micro Instrumentation and Telemetry Systems). The Altair 8080 was one of the first computers available for home use. It came in kit form and had to be assembled before use. In 1974, "kit form" meant that you purchased a machine's individual components, such as transistors, resistors, capacitors, and chips. You would then carefully populate the machine's circuit boards with the components and solder them into place. After your machine was assembled and working, it still had no memory or input/output board—just a panel of switches and lights on the front of the machine. Needless to say, software for this machine was nonexistent.

Allen saw the Altair 8080 on the cover of *Popular Electronics* magazine and showed the magazine to Gates. They both immediately understood that someone would need to develop software for these new machines. Allen called MITS and told them that he and Gates had developed a version of BASIC that would run on their machine. This was untrue—after MITS expressed interest in seeing their version of BASIC, the pair began writing the software. They did not have access to an Altair 8080, so Gates wrote the BASIC code while Allen wrote a simulator for the 8080 that ran on a PDP-10 for testing purposes. The day Allen demonstrated their version of BASIC for MITS was the first time the software ever ran on the actual hardware; astonishingly, it worked perfectly. MITS purchased the rights to the software and Microsoft was born.

EXERCISES

1. The following events might occur in a typical day. Which events would you handle as exceptions and which would you handle as a normal part of your day? Be sure to explain your rationale to classify each event.
   - Your alarm clock going off in the morning
   - A power failure that causes your alarm clock to stop working
   - Taking a shower and running out of hot water
   - Burning your toast
   - Dropping your books on the way to class
   - Missing lunch
   - School is closed because of a health department emergency
   - Running out of quarters at the launderette
2. Find other computer languages that support exceptions. How are exceptions implemented in these languages? How is the implementation different from Java's? Give two advantages and two disadvantages of the way exceptions are implemented in each language when compared with Java.
3. Read the paper cited in footnote 1 of Section 10.1. How closely does Java follow the ideas described in the paper?
4. Which digits print when the following program runs, and in which order?

```
public class MyClass {
  public static void main( String args[] ) {
    int k = 0;

    try {
      int i = 5 / k;
    }
    catch ( ArithmeticException e ) {
      System.out.println( "1" );
    }
    catch ( RuntimeException e ) {
      System.out.println( "2" );
      return;
    }
    catch ( Exception e ) {
      System.out.println( "3" );
    }
```

```
    finally {
      System.out.println( "4" );
    }

    System.out.println( "5" );
  }
}
```

5 Java provides two general categories of exceptions: checked and unchecked. Give two reasons you should throw a checked exception. Give two reasons you might throw an unchecked exception.

6 What would be the programming consequences of making all exceptions in Java checked exceptions?

7 What happens if you do not have a catch block for an exception that might be thrown in a method?

8 Consider the following method:

```
public void increaseSize( double factor ) throws SizeException {
  double increase = size * factor;
  size = size + increase;

  if (factor < 0) {
    throw new SizeException ( "Invalid factor" );
  }
}
```

Write the `SizeException` class. Assume that `size` is a properly defined data member of the class. This method computes an object's size increase and applies the increase to the current size. It should throw an exception if the factor is negative (as this method increases size) and leave the size of the object unchanged. Assuming that the `SizeException` is properly defined, does this code work? Does this code properly protect against an invalid factor? If not, rewrite the code to work properly.

9 Describe the catch or declare policy that is enforced by the Java compiler.

10 Why does the following method defined within a Java class generate a compile-time error?

```
public void fileOperation() {
  try {
    FileReader in;
    in = new FileReader( "xxx.yyy" );
    // code omitted...
  }
```

```
  catch( Exception e ) {
    System.out.println( e.getMessage() );
  }
  catch(FileNotFoundException e) {
    System.out.println( "fileOperation: " + e.getMessage() );
  }
}
```

11 Design a `Queue` class that uses exceptions to handle the situations of trying to put objects into a full queue or trying to access objects from an empty queue. Your class should provide `enqueue()`, `dequeue()`, `front()`, `back()`, and `size()` methods.

12 The home heating case study in Chapter 4 was written without exceptions because we had not yet covered them in the text. Look again at the code in Chapter 4 and identify where exceptions should be used. Rewrite the code to use the exceptions you identified.

13 Write a program that takes the name of a file on the command line and prints the length of the file in bytes.

14 Draw a UML class diagram that shows the inheritance relationships among the following classes: `ClassCastException`, `Exception`, `IOException`, `FileNotFoundException`, `NullPointerException`, and `RuntimeException`.

15 If you have the following declaration in your program, what is the program trying to do? Be sure to specify what kind of data is coming into the program, where it is coming from, any conversions made to it, and what it looks like to the program itself.

```
BufferedReader in =
  new BufferedReader( new InputStreamReader( System.in ) );
```

16 Write a Java program that works like the `FileEcho` program in Figure 10-22, except that it prints the line number in addition to the contents of the line.

17 Write a program called `Replace.java` that takes three command-line arguments: a file name, a search string, and a replace string. The program must open the input file and replace all occurrences of the search string with the replace string. The resulting output is printed to standard output. This type of find-and-replace operation is a common feature of most word-processing programs.

18 Write a processing stream class named `ReplaceStream` that replaces all occurrences of a specified string with a replacement string. The constructor for your stream class should take three arguments: the search string, the replace string, and a `Reader` that contains the text to be changed. Rewrite the program in Exercise 18 so that it uses the `ReplaceStream` class to do the work.

19 The UNIX utility named `wc` reads one or more input files and, by default, writes the number of newline characters, words, and bytes contained in each input file to the standard output. The utility also writes a total count for all named files if more than one input file is specified. The `wc` utility considers a word to be a nonzero-length string of characters delimited by white space (for example, a space or tab). On your favorite UNIX system, look at the main page for `wc`, and then write a Java program that implements "your" version of `wc`.

20 Write a program that reads the contents of a text file and produces a report of the 10 most common and 10 least common words. The program should be written so that the name of the file to process is taken from the command line. To have some fun with your program after it is written, look on the Web for Project Gutenberg, *promo.net/pg/*, which is a project to convert some classic books to electronic form. Run your program using some of these book files.

21 Write a program that takes the names of two or more text files from the command line. The program produces as output a list of the words that are common to all files.

22 A checksum is a simple measure for protecting the integrity of electronic data. Although you can compute a checksum for a file in many ways, they are all similar. For example, you can read the contents of the file a byte at a time (the file data is read as bytes, regardless of whether it is character or byte data) and then compute a running sum of the bytes in the file. Because these sums are large numbers, the sum should be reduced to a fixed size using a modulus. For example, a 16-bit checksum would store the result of adding all the data values in the file as a 16-bit number by computing the sum mod 65536 ($2^{16} = 65536$). Write a program that computes a checksum of an arbitrary file using the technique described in this exercise.

23 Write a Java program that takes the name of a compressed file on the command line and prints the names of the files contained in the archive. Note that several classes in the `java.io` package make this program relatively easy to write.

24 Write a program that computes the number of times certain words appear in a document. Your program works with two files. The first contains the words to be counted. Each line in this file contains a single word. Your program then reads the second file and prints as output the number of times each word in the first file appeared in the second.

25 Virtually every operating system stores directory information in a file. The content and format of these files vary from system to system. Using any resources at your disposal, find out the format of directory files on your system. Using this information, write a Java program that prints the names of the files in a given directory. You should not use the `File` class to write this program.

CHALLENGE WORK EXERCISES

1. A compiler typically consists of a module called a scanner that translates a program from text form to a stream of tokens. A token is the smallest meaningful unit of a program; an identifier is an example of a token in Java. Write a program that uses a `StreamTokenizer` class in the `java.io` package to scan a Java program, converts it into a stream of tokens, and prints all of the declarations that appear in the program.

2. Huffman encoding is a technique you can use to compress data. Using resources at your disposal, look up Huffman encoding and study the algorithms that can compress and decompress data. After you understand the algorithm, write a `HuffmanReader` and a `HuffmanWriter` class that can read and write Huffman encoded files.

3. Many other languages provide support for exception handling. Select a language other than Java and use it to rewrite the exception examples in this chapter. Based on your work, which language do you think has a better syntax for dealing with exceptions? Explain your answer.

4. In Exercise 22, you were asked to write a program that implemented a simple checksum. This checksum would never be used in a system that required a high level of security. The MD5 checksum is an example of a more "industrial"-grade checksum. Rewrite the program from Exercise 22, this time using the more complex MD5 checksum algorithm. Verify that your program works correctly by using available tools on your computer system to compute MD5 checksums. If you do not have such tools on your system, locate a few on the Web and install them on your machine.

5. Look up the format of a Java class file. Write a program that reads a Java class file and prints the following information:

   a. The name of the class

   b. The superclass of the class

   c. The interfaces the class implements (if any)

   d. The classes extended by this class

   e. The access attributes for the class

   f. The names of the methods and fields in the class and their corresponding access attributes