

# Student Workbook for *Murach's Java SE 6*

Introduction for students .....	ii
Chapter 1 .....	1
Chapter 2 .....	6
Chapter 3 .....	14
Chapter 4 .....	22
Chapter 5 .....	29
Chapter 6 .....	38
Chapter 7 .....	47
Chapter 8 .....	56
Chapter 9 .....	65
Chapter 10 .....	73
Chapter 11 .....	83
Chapter 12 .....	91
Chapter 13 .....	99
Chapter 14 .....	107
Chapter 15 .....	115
Chapter 16 .....	121
Chapter 17 .....	129
Chapter 18 .....	137
Chapter 19 .....	143
Chapter 20 .....	155
Chapter 21 .....	163
Chapter 22 .....	172



# Introduction for students

This workbook is designed as a companion to *Murach's Java SE 6*. It provides the study aids that you need for mastering Java. The topics that follow describe the components of this workbook and explain how to get the most from them.

## Behavioral objectives

---

The behavioral objectives for each chapter describe the skills that you should have when you complete the chapter. The educational theory is that if you know what the objectives are, your study will be more focused. In particular, you'll know exactly what to study for the tests because the test questions that we supply only test the skills that are described by the objectives.

If you study the objectives, you can see that the first objectives for each chapter are what we refer to as *applied objectives*. These ask you to apply what you've learned as you develop Java programs. If you can develop the applications that are described by the projects in this workbook, you will meet these objectives.

After the applied objectives for each chapter, you'll find what we refer to as *knowledge objectives*. These objectives define skills like identifying, describing, and explaining the required concepts, terms, and procedures. In general, you should be able to do the knowledge objectives, even if you have trouble with the applied objectives.

In some cases, it makes sense to review the objectives before you read a chapter. That way, you'll know what to focus on. Often, though, the objectives won't make sense until after you read the chapter. That's why we think it's better to review the objectives after you read a chapter. Then, if you don't think you can do one or more of the objectives, you can review the related portions of the chapter.

## Chapter summaries

---

This workbook also includes summaries of the information presented in each chapter. These summaries are bulleted lists of the main concepts and programming techniques, in the same sequence that they are presented in the text. Then, if you aren't clear about one or more of these summary points, you can review the topics that present them. Although these are the same summaries that are presented in the text, we hope you'll agree that it helps to have them in this workbook, right after the objectives.

## Terms

---

The terms list for each chapter contains all of the new terms that are presented in the chapter. After reading a chapter, you can scan the list to make sure you have a general understanding of what each term means. Then, if necessary, you can review the terms you don't understand.

When you use these lists, please keep in mind that you don't need to be able to write definitions of the terms. You should, however, be able to understand them when you hear them in conversations. And you should be able to use the major terms in conversations that you initiate.

## Exercises

---

For each chapter, this workbook presents one or more exercises. Some of these exercises guide you through the development of applications, and some force you to apply what you've learned in new ways. Although these are the same exercises that are in the book, this workbook lets you print them out on separate sheets so they're easier to work with. If you can do all of the exercises, you should be able to meet the knowledge objectives for each chapter.

To help you get the most practice in the least time, you start most of the exercises from code that we provide. We also provide all of the test data that you need for the exercises. Before you start the exercises, then, you need to download these files from our web site and install them on your system. To do that, you can follow the procedures in appendix A of the textbook or go to the Downloads portion of our web site, click on the book title, and download the self-extracting zip file named "All Book Files."

## Projects

---

The projects for each chapter are designed to test your ability to develop Java applications from scratch, and your instructor will most likely assign one or more projects for each section of the book. These projects, of course, are the ultimate test of your Java mastery. If you can do them all, you have developed an impressive set of Java skills.

Although the project specifications give you all of the information that you need for doing each project, you may have to figure out some of the details on your own. That's realistic, though, because most of the specifications in the real world don't provide all of the implementation details. In some cases, then, you may need to ask your instructor questions that clarify details about what the requirements are or how an application should be implemented. In other cases, your instructor may provide details that clarify or enhance an application before you ask.

Here again, we provide all of the source files and data that you need for doing the projects. If you downloaded this workbook from our web site, the required files have already been installed on your system as part of that procedure. Otherwise, you can get them from your instructor.

## Conclusion

---

Since we know how much there is to learn and how little time the modern student has, we have tried to include only those activities that help you master the essential skills in the most efficient way possible. So, after you read each chapter in the book, you can review the objectives, chapter summaries, and terms lists to make sure you understand the critical concepts and terms.

Once you've done that, the exercises and projects will provide the practice you need for mastering Java programming. They will also help you experience the excitement of programming. Although you couldn't possibly do all of the exercises and projects in a single semester, the more you do, the better you're going to get.



# Chapter 1

## How to get started with Java

### Objectives

---

#### Applied

- Install Java SE 6 and the API documentation. If necessary, configure your system to work with the JDK.
- Given the name of a package and a class, look it up in the documentation for the API.
- Given the source code for a Java application, use TextPad to enter, edit, compile, and run the program.
- Given the source file for a Java application, compile and run the program from the command prompt.

#### Knowledge

- Describe how Java compares with C++ and C# based on these features: syntax, platform independence, speed, and memory management.
- Name and describe the three types of programs that you can create with Java.
- Describe how Java compiles and interprets code.
- Explain how the use of bytecodes lets Java achieve platform independence.
- Explain the purpose of setting the Windows command path to work with the JDK.
- Explain what the class path is used for and when you should set it.
- Explain the difference between a compile-time error and a runtime error.
- Describe the benefits of using a Java IDE like Eclipse, NetBeans, or BlueJ.

#### Summary

---

- You use the *Java Development Kit (JDK)* to develop Java programs. This used to be called the *Software Development Kit (SDK)* for Java. As of version 6, the *Standard Edition (SE)* of Java is called *Java SE*. In older versions, it was called the *Java Platform 2, Standard Edition (J2SE)*.
- You can use Java SE to create *applications* and a special type of Internet-based application known as an *applet*. In addition, you can use the *Enterprise Edition (EE)*, which is known as *Java EE*, to create server-side applications using *servlets* and *JavaServer Pages (JSPs)*.
- The *Java compiler* translates *source code* into a *platform-independent* format known as *Java bytecodes*. Then, the *Java interpreter*, or *Java Runtime Environment (JRE)*, translates the bytecodes into instructions that can be run by a specific operating system. A Java interpreter is an implementation of a *Java virtual machine (JVM)*.

- When you use the JDK with Windows, you should add the bin directory (usually C:\Program Files\Java\jdk1.6.0\bin) to the *command path* and you should add the current directory to the *classpath*.
- A *text editor* that's designed for working with Java provides features that make it easier to enter, edit, and save Java code.
- Some text editors such as TextPad include commands for compiling and running Java applications. You can also use the *command prompt* to enter the commands for compiling and running an application.
- When you compile a program, you may get *compile-time errors*. When you run a program, you may get *runtime errors*.
- To compile code from the command prompt, you use the *javac command* to start the Java compiler. To run an application from the command prompt, you use the *java command* to start the Java interpreter.
- You can get detailed information about any class in the J2SE by using a web browser to browse the HTML-based documentation for its *Application Programming Interface (API)*.
- An *Integrated Development Environment (IDE)* like Eclipse, NetBeans, or BlueJ can make working with Java easier.

## Terms

---

Java Development Kit (JDK)	text editor
Software Development Kit (SDK)	case-sensitive language
Java Standard Edition (SE)	ASCII format
application	ANSI format
graphical user interface (GUI)	console
applet	compile-time error
servlet	runtime error
JavaServer Pages (JSPs)	public class
class	command prompt
source code	DOS prompt
Java compiler	DOS window
bytecodes	javac command
Java interpreter	java command
interpret	switch
platform independence	deprecated features
Java virtual machine (JVM)	Application Programming Interface (API)
Java Plug-in	package
Java Runtime Environment (JRE)	Integrated Development Environment (IDE)
command path	
autoexec.bat file	
class path	

## Exercise 1-1 Use TextPad to develop an application

---

This exercise will guide you through the process of using TextPad to enter, save, compile, and run a simple application.

### Enter and save the source code

1. Start TextPad by clicking on the Start button and selecting Programs or All Programs→TextPad.
2. Enter this code (type carefully and use the same capitalization):

```
public class TestApp
{
    public static void main(String[] args)
    {
        System.out.println(
            "This Java application has run successfully");
    }
}
```

3. Use the Save command in the File menu to display the Save As dialog box. Next, navigate to the c:\java1.6\ch01 directory and enter TestApp in the File name box. If necessary, select the Java option from the Save as Type combo box. Then, click on the Save button to save the file.

### Compile the source code and run the application

4. Press Ctrl+1 to compile the source code. If you get an error message, read the error message, edit the text file, save your changes, and compile the application again. Repeat this process until you get a clean compile.
5. Press Ctrl+2 to run the application. This application should display a console window that says “This Java application has run successfully” followed by a line that reads “Press any key to continue...”.
6. Press any key. This should close the console window. If it doesn't, click on the Close button in the upper right corner of the window to close it.

### Introduce and correct a compile-time error

7. In the TextPad window, delete the semicolon at the end of the System.out.println statement. Then, press Ctrl+1 to compile the source code. TextPad should display an error message that indicates that the semicolon is missing in the Command Results window.
8. In the Document Selector pane, click on the TestApp.java file to switch back to the source code, and press Ctrl+F6 twice to toggle back and forth between the Command Result window and the source code. Then, select View→Line Numbers to display the line numbers for the source code lines.
9. Correct the error and compile the file again (this automatically saves your changes). This time the file should compile cleanly, so you can run it again and make sure that it works correctly.
10. Select Configure→Preferences, click on View, and check Line Numbers. That will add line numbers to the source statements in all your applications. If you want to look through the other options and set any of them, do that now. When you're done, close the file and exit TextPad.

## Exercise 1-2 Use any Java development tool to develop an application

---

If you aren't going to use TextPad to develop your Java programs, you can try whatever tools you are going to use with this generic exercise.

### Use any text editor to enter and save the source code

1. Start the text editor and enter this code (type carefully and use the same capitalization):

```
public class TestApp
{
    public static void main(String[] args)
    {
        System.out.println(
            "This Java application has run successfully");
    }
}
```

2. Save this code in the c:\java1.6\ch01 directory in a file named "TestApp.java".

### Compile the source code and run the application

3. Compile the source code. If you're using a text editor that has a compile command, use this command. Otherwise, use your command prompt to compile the source code. To do that, start your command prompt and use the cd command to change to the c:\java1.6\ch01 directory. Then, enter the javac command like this (make sure to use the same capitalization):

```
javac TestApp.java
```

4. Run the application. If you're using a text editor that has a run or execute command, use this command. Otherwise, use your command prompt to run the application. To do that, enter the java command like this (make sure to use the same capitalization):

```
java TestApp
```

5. When you enter this command, the application should print "This Java application has run successfully" to the console window.

## Exercise 1-3 Use the command prompt to run any compiled application

---

This exercise shows how to use the command prompt to run any Java application.

1. Open the command prompt window. Then, change the current directory to c:\java1.6\ch01.
2. Use the java command to run the LoanCalculatorApp application. This application calculates the monthly payment for a loan amount at the interest rate and number of years that you specify. This shows how the JRE can run any application whether or not it has been compiled on that machine. When you're done, close the application to return to the command prompt.



## **Exercise 1-4      Navigate the API documentation**

---

This exercise will give you some practice using the API documentation to look up information about a class.

1. Start a web browser and navigate to the index page that contains the API documentation for the JDK (usually `C:\Program Files\Java\jdk1.6.0\docs\api\index.htm`). This page should look like the one shown in figure 1-16.
2. Bookmark this page so you can easily access it later. To do that with the Internet Explorer, select the Add To Favorites item from the Favorites menu. Then, close your web browser.
3. Start your web browser again and use the bookmark to return to the API documentation for the JDK. To do that with the Internet Explorer, select the Java 2 Platform SE item from the Favorites menu.
4. Select the `java.lang` package in the upper left frame and notice that the links in the lower left frame change. Select the `System` class from this frame to display information about it in the right frame.
5. Scroll down to the Field Summary area in the right frame and click on the out link for the standard output stream. When you do, an HTML page that gives some information about how to use the standard output stream will be displayed. In the next chapter, you'll learn more about using the out field of the `System` class to print data to the console.
6. Continue experimenting with the documentation until you're comfortable with how it works. Then, close the browser.

## Chapter 2

# Introduction to Java programming

### Objectives

---

#### Applied

- Given the specifications for an application that requires only the language elements presented in this chapter, write, test, and debug the application.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- Explain how the two types of comments are typically used in a Java application.
- Given a list of names, identify the ones that are valid for Java classes and variables.
- Given a list of names, identify the ones that follow the naming recommendations for classes presented in this chapter.
- Given a list of names, identify the ones that follow the naming recommendations for variables presented in this chapter.
- Describe the difference between a main method and other methods.
- Name three things you can assign to a numeric variable.
- Distinguish between the int and double data types.
- Explain what happens when an arithmetic expression uses both int and double values.
- Name three things you can assign to a String variable.
- Explain what an escape sequence is and when you would use one.
- Explain what *importing a class* means and when you typically do that.
- Explain what a static method is and how it differs from other methods.
- Explain what a Scanner object can be used for.
- Explain what the System.out object can be used for.
- Explain what a Boolean expression is and when you might use one.
- Explain what it means for a variable to have block scope.
- Describe the difference between testing an application and debugging an application.
- Describe the difference between a runtime error and a logical error.

## Summary

---

- The *statements* in a Java program direct the operation of the program. The *comments* document what the program does.
- You must code at least one public *class* for every Java program that you write. The *main method* of this class is executed when you run the class.
- *Variables* are used to store data that changes as a program runs, and you use *assignment statements* to assign values to variables. Two of the most common *data types* for numeric variables are the `int` and `double` types.
- A *string* is an object that's created from the `String` class, and it can contain any characters in the character set. You can use the plus sign to *join* a string with another string or a data type, and you can use assignment statements to *append* one string to another. To include special characters in strings, you can use *escape sequences*.
- Before you use many of the classes in the Java API, you should code an import statement for the class or for the *package* that contains it.
- When you use a *constructor* to create an *object* from a Java class, you are creating an *instance* of the class. There may be more than one constructor for a class, and a constructor may require one or more *arguments*.
- You *call a method* from an object and you call a *static method* from a class. A method may require one or more arguments.
- One of the most time-consuming aspects of Java programming is researching the classes and methods that your programs require.
- You can use the methods of a `Scanner` object to read data from the *console*, and you can use the `print` and `println` methods of the `System.out` object to print data to the console.
- You can code *if statements* to control the logic of a program based on the true or false values of *Boolean expressions*. You can code *while statements* to repeat a series of statements until a Boolean expression becomes false.
- *Testing* is the process of finding the errors or bugs in an application. *Debugging* is the process of fixing the bugs.

## Terms

---

statement	package
block of code	Abstract Window Toolkit (AWT)
comment	Swing
single-line comment	object
end-of-line comment	constructor
block comment	argument
identifier	instance
keyword	instantiation
class	calling a method
class declaration	static method
access modifier	console
scope	token
method	whitespace
main method	exception
main method declaration	control statement
method argument	Boolean expression
variable	Boolean value
primitive data type	relational operator
integer	if/else statement
initialize a variable	if statement
declare a data type	selection structure
camel notation	block scope
literal	nested if statements
assignment statement	while statement
arithmetic expression	iteration structure
arithmetic operator	while loop
operand	counter variable
casting	counter
string	infinite loop
string literal	test
empty string	bug
null value	runtime error
join	runtime exception
concatenate	logical error
append	debug
escape sequence	

## **Exercise 2-1      Test the Invoice application**

---

In this exercise, you'll compile and test the Invoice application that's presented in figure 2-18. That will give you a better idea of how this program works.

1. Start your text editor and open the file named `InvoiceApp.java` that you should find in the `c:\java1.6\ch02` directory. Then, compile the application, which should compile with no errors.
2. Test this application with valid subtotal entries like 50, 150, 250, and 1000 so it's easy to see whether or not the calculations are correct.
3. Test the application with a subtotal value like 233.33. This will show that the application doesn't round the results to two decimal places. But in the next chapter, you'll learn how to do that.
4. Test the application with an invalid subtotal value like \$1000. This time, the application should crash. Study the error message that's displayed and determine which line of source code was running when the error occurred.
5. Restart the application, enter a valid subtotal, and enter 20 when the program asks you whether you want to continue. What happens and why?
6. Restart the application and enter two values separated by whitespace (like 1000 20) before pressing the Enter key. What happens and why?

## **Exercise 2-2      Modify the Test Score application**

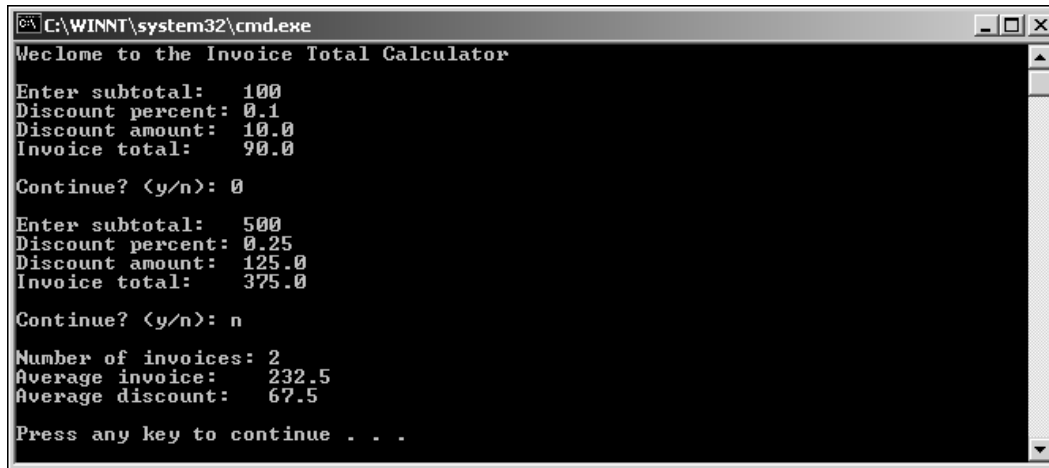
---

In this exercise, you'll modify the Test Score application that's presented in figure 2-19. That will give you a chance to write some code of your own.

1. Open the file named `TestScoreApp.java` in the `c:\java1.6\ch02` directory, and save the program as `ModifiedTestScoreApp.java` in the same directory. Then, change the class name to `ModifiedTestScoreApp` and compile the class.
2. Test this application with valid data to see how it works. Then, test the application with invalid data to see what will cause exceptions. Note that if you enter a test score like 125, the program ends, even though the instructions say that the program ends when you enter 999.
3. Modify the while statement so the program only ends when you enter 999. Then, test the program to see how this works.
4. Modify the if statement so it displays an error message like "Invalid entry, not counted" if the user enters a score that's greater than 100 but isn't 999. Then, test this change.

## Exercise 2-3      Modify the Invoice application

In this exercise, you'll modify the Invoice application. When you're through with the modifications, a test run should look something like this:



```
C:\WINNT\system32\cmd.exe
Welcome to the Invoice Total Calculator
Enter subtotal: 100
Discount percent: 0.1
Discount amount: 10.0
Invoice total: 90.0
Continue? <y/n>: 0
Enter subtotal: 500
Discount percent: 0.25
Discount amount: 125.0
Invoice total: 375.0
Continue? <y/n>: n
Number of invoices: 2
Average invoice: 232.5
Average discount: 67.5
Press any key to continue . . .
```

1. Open the file named `InvoiceApp.java` that's in the `c:\java1.6\ch02` directory, and save the program as `ModifiedInvoiceApp.java` in the same directory. Then, change the class name to `ModifiedInvoiceApp`.
2. Modify the code so the application ends only when the user enters "n" or "N". As it is now, the application ends when the user enters anything other than "y" or "Y". To do this, you need to use a not operator (!) with the `equalsIgnoreCase` method. This is illustrated by the third example in figure 2-15. Then, compile this class and test this change by entering 0 at the `Continue?` prompt.
3. Modify the code so it provides a discount of 25 percent when the subtotal is greater than or equal to \$500. Then, test this change.
4. Using the `Test Score` application as a model, modify the `Invoice` program so it displays the number of invoices, the average invoice amount, and the average discount amount when the user ends the program. Then, test this change.

## Exercise 2-4      Use the Java API documentation

---

This exercise steps you through the Java API documentation for the Scanner, String, and Double classes. That will give you a better idea of how extensive the Java API is.

1. Go to the index page of the Java API documentation as described in chapter 1. If you did the exercises for that chapter, you should have it bookmarked.
2. Click the `java.util` package in the upper left window and the Scanner class in the lower left window to display the documentation for the Scanner class. Then, scroll through this documentation to get an idea of its scope.
3. Review the constructors for the Scanner class. The constructor that's presented in this chapter has just an `InputStream` object as its argument. When you code that argument, remember that `System.in` represents the `InputStream` object for the console.
4. Review the methods of the Scanner class with special attention to the `next`, `nextInt`, and `nextDouble` methods. Note that there are three `next` methods and two `nextInt` methods. The ones used in this chapter have no arguments. Then, review the `has` methods in the Scanner class. You'll learn how to use some of these in chapter 5.
5. Go to the documentation for the String class, which is in the `java.lang` package, and note that it offers a number of constructors. In this chapter, though, you learned the shortcut for creating String objects because that's the best way to do that. Now, review the methods for this class with special attention to the `equals` and `equalsIgnoreCase` methods.
6. Go to the documentation for the Double class, which is also in the `java.lang` package. Then, review the static `parseDouble` and `toString` methods that you'll learn how to use in the next chapter.

If you find the documentation difficult to follow, rest assured that you'll become comfortable with it before you finish this book. Once you learn how to create your own classes, constructors, and methods, it will make more sense.

---

## Project 2-1: Calculate a rectangle's area and perimeter

---

### Console

```
Welcome to the Area and Perimeter Calculator

Enter length: 100
Enter width: 200
Area: 20000.0
Perimeter: 600.0

Continue? (y/n): y

Enter length: 8
Enter width: 4
Area: 32.0
Perimeter: 24.0

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter values for the length and width of a rectangle.
- The application displays the area and perimeter of the rectangle.
- The application prompts the user to continue.

### Specifications

- The formulas for calculating area and perimeter are:  
`area = width * length`  
`perimeter = 2 * width + 2 * length`
- The application should accept decimal entries like 10.5 and 20.65.
- Assume that the user will enter valid numeric data for the length and width.
- The application should continue only if the user enters “y” or “Y” to continue.



---

## Project 2-2: Convert number grades to letter grades

---

### Console

```
Welcome to the Letter Grade Converter

Enter numerical grade: 90
Letter grade: A

Continue? (y/n): y

Enter numerical grade: 88
Letter grade: A

Continue? (y/n): y

Enter numerical grade: 80
Letter grade: B

Continue? (y/n): y

Enter numerical grade: 67
Letter grade: C

Continue? (y/n): y

Enter numerical grade: 59
Letter grade: F

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The user enters a numerical grade from 0 to 100.
- The application displays the corresponding letter grade.
- The application prompts the user to continue.

### Specifications

- The grading criteria is as follows:

A	88-100
B	80-87
C	67-79
D	60-67
F	<60
- Assume that the user will enter valid integers for the grades.
- The application should continue only if the user enters “y” or “Y” to continue.

## Chapter 3

# How to work with data

### Objectives

---

#### Applied

- Given the specifications for an application that uses any of the eight primitive data types presented in this chapter, write the application.
- Use the `NumberFormat`, `Math`, `Integer`, `Double`, and `BigDecimal` classes to work with data.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- Describe any one of the eight primitive types.
- Distinguish between a variable and a constant.
- Given a list of names, identify the ones that follow the naming recommendations for constants presented in this chapter.
- Explain the difference between a binary operator and a unary operator and give an example of each.
- Explain the difference between prefixing and postfixing an increment or decrement operator.
- Explain what a shortcut assignment operator is and how you use one in an assignment statement.
- List the order of precedence for arithmetic operations and explain how you can change the order in which operations are performed.
- Explain what casting is, when it's performed implicitly, and when you must perform it explicitly.
- Describe how casting between `int` and `double` types can affect the decimal value in a result.
- Describe the primary uses of these classes: `NumberFormat`, `Math`, `Integer`, and `Double`.
- List two reasons for using the `BigDecimal` class.

## Summary

---

- Java provides eight *primitive data types* to store *integer*, *floating-point*, *character*, and *boolean* values.
- *Variables* store data that changes as a program runs. *Constants* store data that doesn't change as a program runs. You use *assignment statements* to assign values to variables.
- You can use *arithmetic operators* to form *arithmetic expressions*, and you can use some *assignment operators* as a shorthand for some types of arithmetic expressions.
- Java can *implicitly cast* a less precise data type to a more precise data type. Java also lets you *explicitly cast* a more precise data type to a less precise data type.
- You can use the `NumberFormat` class to apply standard currency, percent, and number formats to any of the primitive numeric types.
- You can use the static methods of the `Math` class to perform mathematical operations such as rounding numbers and calculating square roots.
- You can use the constructors of the `Double` and `Integer` *wrapper classes* to create objects that wrap double and int values. You can also use the static methods of these classes to convert strings to numbers and vice versa.
- You can use the constructors of the `BigDecimal` class to create objects that store decimal values that aren't limited to 16 significant digits. Then, you can use the methods of these objects to do the calculations that your programs require.

## Terms

---

primitive data type	operand
bit	literal
byte	binary operator
integer	unary operator
floating-point number	assignment statement
significant digit	prefixed operator
single precision number	postfixed operator
double precision number	assignment operator
scientific notation	order of precedence
Unicode character set	casting
ASCII character set	implicit cast
boolean	widening conversion
variable	narrowing conversion
constant	explicit cast
declare	half-even
initialize	wrapper class
assign an initial value	throw an exception
final variable	catch an exception
arithmetic expression	debugging statement
arithmetic operator	scale

## Exercise 3-1 Test the Invoice application

---

In this exercise, you'll compile and test the formatted Invoice application that's presented in figure 3-10.

1. Open the file named `FormattedInvoiceApp.java` that you should find in the `c:\java1.6\ch03` directory. Then, compile and run the application. As you test the application, enter the three subtotal values that are shown in figures 3-10 and 3-11 to see how the program works and to see what the problems are.
2. To better understand what is happening, add debugging statements like those in figure 3-11 so the program displays two sets of data for each entry: first the unformatted output, then the formatted output. When you add debugging statements, you should try to do it in a way that makes them easy to remove when you're through debugging.
3. Test the application again with a range of entries so you clearly see what the data problems are when you study the unformatted and formatted results.

## Exercise 3-2 Modify the Test Score application

---

In this exercise, you'll use some of the skills that you learned in this chapter as you modify the Test Score application that you worked with in the last chapter, but you won't use `BigDecimal` arithmetic.

1. Open the file named `ModfiedTestScoreApp.java` that you should find in the `c:\java1.6\ch02` directory if you did exercise 2-2. If you didn't do that exercise, open `TestScoreApp` instead.
2. Save the file as `EnhancedTestScoreApp.java` in the `ch03` directory, and change the class name in the file to `EnhancedTestScoreApp`. Then, compile and run the program to refresh your memory about how it works.
3. Use the `+=` operator to increase the `scoreCount` and `scoreTotal` fields. Then, test this to make sure that it works.
4. As the user enters test scores, use the methods of the `Math` class to keep track of the minimum and maximum scores. When the user enters 999 to end the program, display these scores at the end of the other output data. Now, test these changes to make sure that they work. (This step can be challenging if you're new to programming, but you'll learn a lot by doing it.)
5. Change the variable that you use to total the scores from a `double` to an `int` data type. Then, use casting to cast the score count and score total to `doubles` as you calculate the average score and save that average as a `double`. Now, test that change.
6. Use the `NumberFormat` class to round the average score to one decimal place before displaying it at the end of the program. Then, test this change. Note that the rounding method that's used doesn't matter in a program like this.

### Exercise 3-3 Create a new application

---

In this exercise, you'll develop an application that will give you a chance to use your new skills. This application asks the user to enter a file size in megabytes (MB) and then calculates how long it takes to download that file with a 56K analog modem (you won't need to use BigDecimal arithmetic). The output from this application should look something like this:

```
Welcome to the Download Time Estimator

This program calculates how long it will take to
download a file with a 56K analog modem.

Enter file size (MB): 50

A "56K" modem will take 2 hours 44 minutes 6 seconds

Continue? (y/n):
```

1. Instead of starting this application from scratch, open the file named `FormattedInvoiceApp.java` in the `ch03` directory. Then, save it with the name `DownloadTimeApp.java`, and change its class name to `DownloadTimeApp`.
2. Delete the code that you won't need for this application, and modify the code that remains so it provides for the basic operation of the program without the calculations. These first two steps are an efficient way to start any new application because you don't have to re-enter the routine code.
3. Add the code that calculates the hours, minutes, and seconds needed to download this file with a 56K analog modem. To do the calculations, assume that a 56K modem can transfer data at the rate of 5.2 kilobytes (KB) per second. Then, add the code for displaying the results. (You also need to know that 1 MB is equal to 1,024 KB).
4. Compile and run the application. Enter a value of 50 for the file size to be sure that the calculated value is the same as shown above. Then, enter other values to see how they work.

### Exercise 3-4 Use BigDecimal arithmetic

---

To get some practice with BigDecimal arithmetic, this exercise has you modify the Test Score application so it uses BigDecimal arithmetic.

1. Open `EnhancedTestScoreApp` in the `ch03` directory or your last version of the Test Score application in the `ch02` directory. Then, save the file as `BDTestScoreApp` in `ch03`, and change the class name to `BDTestScoreApp`.
2. Modify the program so it uses BigDecimal arithmetic to calculate the average test score with the result rounded to one decimal place. Be sure to use the appropriate `toString` methods of either the `Double` or `Integer` classes so the `BigDecimal` objects are constructed from string values. Then, test this change with a range of values.

---

## Project 3-1: Convert temperature from Fahrenheit to Celsius

---

### Console

```
Welcome to the Temperature Converter

Enter degrees in Fahrenheit: 212
Degrees in Celsius: 100

Continue? (y/n): y

Enter degrees in Fahrenheit: 32
Degrees in Celsius: 0

Continue? (y/n): y

Enter degrees in Fahrenheit: 77.5
Degrees in Celsius: 25.28

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter a temperature in Fahrenheit degrees.
- The application displays the temperature in Celsius degrees.
- The application prompts the user to continue.

### Specifications

- The formula for converting temperatures from Fahrenheit to Celsius is:  
$$c = (f - 32) * 5/9$$
- The application should accept decimal entries like 77.5.
- Assume that the user will enter valid data.
- The application should continue only if the user enters “y” or “Y” to continue.

---

## Project 3-2: Calculate travel time based on distance and speed

---

### Console

```
Welcome to the Travel Time Calculator

Enter miles:          200
Enter miles per hour: 65

Estimated travel time
Hours:   3
Minutes: 4

Continue? (y/n): y

Enter miles:          100
Enter miles per hour: 65

Estimated travel time
Hours:   1
Minutes: 32

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter values for miles and miles per hour.
- The application displays the approximate travel time in hours and minutes.
- The application prompts the user to continue.

### Specifications

- The application should accept decimal entries like 10.5 and 20.65.
- Assume that the user will enter valid data.
- The application should continue only if the user enters “y” or “Y” to continue.

### Hint

- Use integer arithmetic and the division and modulus operators to get hours and minutes.

---

## Project 3-3: Calculate interest

---

### Console

```
Welcome to the Interest Calculator

Enter loan amount: 520000
Enter interest rate: .05375

Loan amount:          $520,000.00
Interest rate:        5.375%
Interest:              $27,950.00

Continue? (y/n): y

Enter loan amount: 4944.5
Enter interest rate: .01

Loan amount:          $4,944.50
Interest rate:        1%
Interest:              $49.45

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter a loan amount and an interest rate.
- The application calculates the interest amount and formats the loan amount, interest rate, and interest amount. Then, it displays the formatted results to the user.
- The application prompts the user to continue.

### Specifications

- This application should use the `BigDecimal` class to make sure that all calculations are accurate. It should round the interest that's calculated to two decimal places, rounding up if the third decimal place is five or greater.
- The value for the formatted interest rate should allow for up to 3 decimal places.
- Assume that the user will enter valid double values for the loan amount and interest rate.
- The application should continue only if the user enters "y" or "Y" to continue.



---

## Project 3-4: Calculate coins for change

---

### Console

```
Welcome to the Change Calculator

Enter number of cents (0-99): 99

Quarters: 3
Dimes:    2
Nickels:  0
Pennies:  4

Continue? (y/n): y

Enter number of cents (0-99): 55

Quarters: 2
Dimes:    0
Nickels:  1
Pennies:  0

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter a number of cents from 0 to 99.
- The application displays the minimum number of quarters, dimes, nickels, and pennies that represent the coins that make up the specified number of cents.
- The application prompts the user to continue.

### Specifications

- Assume that the user will enter a valid integer value for the number of cents.
- The application should continue only if the user enters “y” or “Y” to continue.

## Chapter 4

# How to code control statements

### Objectives

---

#### Applied

- Code if/else statements and switch statements to control the logic of an application.
- Code while, do-while, and for loops to control the repetitive processing that an application requires.
- Code nested for loops whenever they are required.
- Use the break, labeled break, continue, and labeled continue statements to jump out of a loop or to jump to the start of a loop.
- Code a static method that performs a given operation, and code a statement that calls that method.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- Explain how a reference type like a String object is different from a primitive data type.
- Explain what a logical operator is and why you would use one.
- Describe the difference between how the regular logical operators and the short-circuit operators work.
- Compare the if/else and switch statements.
- Explain what it means for execution to fall through a label in a switch statement.
- Describe the differences between while, do-while, and for loops.
- Explain what an infinite loop is and what you must do when one occurs.
- Describe the difference between the break statement and the continue statement.
- Explain what an access modifier is and how it affects the static methods you define.
- Explain what the signature of a method is.

## Summary

---

- You can use the *relational operators* to create *Boolean expressions* that compare primitive data types and return true or false values, and you can use the *logical operators* to connect two or more Boolean expressions.
- To determine whether two strings are equal, you can call the `equals` and `equalsIgnoreCase` methods from a `String` object.
- You can use *if/else statements* and *switch statements* to control the logic of an application, and you can *nest* these statements whenever necessary.
- You can use *while*, *do-while*, and *for loops* to repeatedly execute one or more statements until a Boolean expression evaluates to false, and you can nest these statements whenever necessary.
- You can use *break statements* to jump to the end of the current loop or a labeled loop, and you can use *continue statements* to jump to the start of the current loop or a labeled loop.
- To code a *static method*, you code an access modifier, the static keyword, its return type, its name, and a *parameter list*. Then, to return a value, you code a *return statement* within the method.
- To call a static method that's in the same class as the main method, you code the method name followed by an *argument list*.

## Terms

---

Boolean expression	while statement
relational operator	while loop
reference type	do-while loop
logical operator	counter variable
short-circuit operator	infinite loop
if/else statement	for loop
if statement	label
selection structure	static method
block of statements	access modifier
block scope	parameter list
nested statements	parameter
switch statement	return statement
case structure	signature
case label	overloading a method
break statement	calling a static method
fall through	argument
default label	argument list
iteration structure	

## **Exercise 4-1      Test the Future Value application**

---

In this exercise, you'll test the Future Value application that's presented in figure 4-9 in this chapter.

1. Open the FutureValueApp class stored in the ch04 directory. Then, compile and test it with valid data to see how it works.
2. To make sure that the results are correct, add a debugging statement within the for loop that calculates the future value. This statement should display the month and future value each time through the loop. Then, test the program with simple entries like 100 for monthly investment, 12 for yearly interest (because that's 1 percent each month), and 1 for year. When the debugging data is displayed, check the results manually to make sure they're correct.

## **Exercise 4-2      Enhance the Invoice application**

---

In this exercise, you'll modify the nested if/else statements that are used to determine the discount percent for the Invoice application in figure 4-6. Then, you'll code and call a static method that determines the discount percent.

### **Open the application and change the if/else statement**

1. Open the application named CodedInvoiceApp that's in the ch04 directory, save it as EnhancedInvoiceApp, and change the class name. Then, compile and run the application to see how it works.
2. Change the if/else statement so customers of type "R" with a subtotal that is greater than or equal to \$250 but less than \$500 get a 25% discount and those with a subtotal of \$500 or more get a 30% discount. Next, change the if/else statement so customers of type "C" always get a 20% discount. Then, test the application to make sure this works.
3. Add another customer type to the if/else statement so customers of type "T" get a 40% discount for subtotals of less than \$500, and a 50% discount for subtotals of \$500 or more. Then, test the application.
4. Check your code to make sure that no discount is provided for a customer type code that isn't "R", "C", or "T". Then, fix this if necessary.

### **Code and call a static method that determines the discount percent**

5. Code a static method named getDiscountPercent that has two parameters: customer type and subtotal. To do that efficiently, you can move the appropriate code from the main method of the application into the static method and make the required modifications.
6. Add code that calls the static method from the body of the application. Then, test to make sure that it works.

## Exercise 4-3 Enhance the Test Score application

In this exercise, you'll enhance the Test Score application so it uses a while or a do-while loop plus a for loop. After the enhancements, the console for a user's session should look something like this:

```
Enter the number of test scores to be entered: 5

Enter score 1: 75
Enter score 2: 80
Enter score 3: 75
Enter score 4: 880
Invalid entry, not counted
Enter score 4: 80
Enter score 5: 95

Score count:    5
Score total:    405
Average score:  81
Minimum score:  75
Maximum score:  95

Enter more test scores? (y/n): y

Enter the number of test scores to be entered: 3

Enter score 1: 85
Enter score 2: 95
Enter score 3: 100

Score count:    3
Score total:    280
Average score:  93.3
Minimum score:  85
Maximum score:  100

Enter more test scores? (y/n):
```

1. Open the EnhancedTestScoreApp in ch03 that you developed for exercise 3-2. If you didn't do that exercise, you can work from an earlier version like the ModifiedTestScoreApp or the TestScoreApp that's in ch02. Then, save the application in the ch04 directory as EnhancedTestScoreApp, change the class name if necessary, and compile and test the application.
2. Change the while statement to a do-while statement, and test this change. Does this work any better than the while loop?
3. Enhance the program so it uses a while or do-while loop that controls whether the user enters more than one set of test scores. The statements in this loop should first ask the user how many test scores are going to be entered. Then, this number should be used in a for loop that gets that many test score entries from the user. When the loop ends, the program should display the summary data for the test scores, determine whether the user wants to enter another set of scores, and repeat the while loop if necessary. After you make these enhancements, test them to make sure they work.
4. If you didn't already do it, make sure that the code in the for loop doesn't count an invalid entry. In that case, an error message should be displayed and the counter decremented by one. Now, test to make sure this works.

---

## Project 4-1: Display a table of powers

---

### Console

```
Welcome to the Squares and Cubes table
```

```
Enter an integer: 9
```

```
Number  Squared  Cubed
=====  =====  =====
1         1         1
2         4         8
3         9        27
4        16        64
5        25       125
6        36       216
7        49       343
8        64       512
9        81       729
```

```
Continue? (y/n): y
```

```
Enter an integer: 3
```

```
Number  Squared  Cubed
=====  =====  =====
1         1         1
2         4         8
3         9        27
```

```
Continue? (y/n): n
```

```
Press any key to continue . . .
```

### Operation

- The application prompts the user to enter an integer.
- The application displays a table of squares and cubes from 1 to the value entered by the user.
- The application prompts the user to continue.

### Specifications

- The formulas for calculating squares and cubes are:

```
square = x * x
cube = x * x * x
```

- Assume that the user will enter a valid integer.
- The application should continue only if the user enters “y” or “Y” to continue.

---

## Project 4-2: Calculate the factorial of a number

---

### Console

```
Welcome to the Factorial Calculator

Enter an integer that's greater than 0 and less than 10: 3
The factorial of 3 is 6.

Continue? (y/n): y

Enter an integer that's greater than 0 and less than 10: 4
The factorial of 4 is 24.

Continue? (y/n): y

Enter an integer that's greater than 0 and less than 10: 9
The factorial of 9 is 362880.

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter an integer from 1 to 9.
- The application displays the factorial of the number entered by the user.
- The application prompts the user to continue.

### Specifications

- The exclamation point is used to identify a factorial. For example, the factorial of the number  $n$  is denoted by  $n!$ . Here's how you calculate the factorial of the numbers 1 through 5:

```
1! = 1                which equals 1
2! = 1 * 2            which equals 2
3! = 1 * 2 * 3        which equals 6
4! = 1 * 2 * 3 * 4    which equals 24
5! = 1 * 2 * 3 * 4 * 5 which equals 120
```

- Use a for loop to calculate the factorial.
- Assume that the user will enter valid numeric data for the length and width.
- Use the long type to store the factorial.
- The application should continue only if the user enters “y” or “Y” to continue.

### Enhancement

- Test the application and find the integer for the highest factorial that can be accurately calculated by this application. Then, modify the prompt so it prompts the user for a number from 1 to the highest integer that returns an accurate factorial calculation.

---

## Project 4-3: Find the greatest common divisor of two positive integers

---

### Console

```
Greatest Common Divisor Finder

Enter first number: 12
Enter second number: 8
Greatest common divisor: 4

Continue? (y/n): y

Enter first number: 77
Enter second number: 33
Greatest common divisor: 11

Continue? (y/n): y

Enter first number: 441
Enter second number: 252
Greatest common divisor: 63

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter two numbers.
- The application displays the greatest common divisor of the two numbers.
- The application prompts the user to continue.

### Specifications

- The formula for finding the greatest common divisor of two positive integers  $x$  and  $y$  follows the Euclidean algorithm as follows:
  1. Subtract  $x$  from  $y$  repeatedly until  $y < x$ .
  2. Swap the values of  $x$  and  $y$ .
  3. Repeat steps 1 and 2 until  $x = 0$ .
  4.  $y$  is the greatest common divisor of the two numbers.
- Place the calculation for finding the divisor in a static method. You can use one loop for step 1 of the algorithm nested within a second loop for step 3.
- Assume that the user will enter valid integers for both numbers.
- The application should continue only if the user enters “y” or “Y” to continue.



## Chapter 5

# How to validate input data

### Objectives

---

#### Applied

- Given an application that uses the console to get input from the user, write code that handles any exceptions that might occur.
- Given an application that uses the console to get input from the user and the validation specifications for that data, write code that validates the user entries.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.
- Given the output for an unhandled exception, determine the cause of the exception.

#### Knowledge

- Explain what an exception is in Java.
- Describe the Exception hierarchy and name two of its subclasses.
- Explain what the stack trace is and how you can use it to determine the cause of an exception.
- Explain how you use the try statement to catch an exception.
- Explain what an exception handler is and when the code in an exception handler is executed.
- Explain how you can use methods of the Scanner class to validate data.
- Explain why it's usually better to validate user entries than to catch and handle exceptions caused by invalid entries.
- Describe the two types of data validation that you're most likely to perform on a user entry.
- Explain why you might want to use generic methods for data validation.

## Summary

---

- An *exception* is an object that's created from the `Exception` class or one of its *subclasses*. This object contains information about an error that has occurred.
- The *stack trace* is a list of methods that were called before an exception occurred.
- You can code a *try statement* to create an *exception handler* that will *catch* and handle any exceptions that are *thrown*. This is known as *exception handling*.
- *Data validation* refers to the process of checking input data to make sure that it's valid.
- *Range checking* refers to the process of checking an entry to make sure that it falls within a certain range of values.

## Terms

---

exception  
exception handling  
throw an exception  
subclass  
catch an exception  
stack trace  
try statement

try/catch statement  
try block  
catch block  
exception handler  
data validation  
range checking

## Exercise 5-1      Add validation to the Invoice application

---

In this exercise, you'll add data validation to the Invoice application that you've worked on in previous chapters. To do that, you'll use exception handling. You'll also write and use some specific data validation methods.

1. Open the `EnhancedInvoiceApp` in the `ch04` directory that you developed for exercise 4-2. If you didn't do that exercise, open `CodedInvoiceApp` in the `ch04` directory. Either way, save the application as `ValidatedInvoiceApp` in the `ch05` directory, change the class name, and compile and run the application.
2. As you test the application, enter an invalid customer type code to see what happens. Then, enter an invalid subtotal entry like \$1000 to see what happens when the application crashes.

### Validate the customer type code

3. Modify the application so it will only accept these customer type codes: `r/c/t` or just `r/c`, depending on which version of the program you're modifying. It should also discard any extra entries on the customer type line. If the user enters an invalid code, the application should display an error message and ask the user to enter a valid code. Now, test this enhancement.
4. Code a static method named `getValidCustomerType` that does the validation of step 3. This method should require one parameter that receives a `Scanner` object, and it should return a valid customer type code. The method should get an entry from the user, check it for validity, display an error message if it's invalid, and discard any other user entries whether or not the entry is valid. This method should continue getting user entries until one is valid.
5. After you've written this method, modify the application so it uses this method. Then, test this enhancement.

### Validate the subtotal

6. Add a try statement so it catches any `InputMismatchException` that the `nextDouble` method of the `Scanner` class might throw. The catch block should display an error message and issue a `continue` statement to jump to the beginning of the while loop. It should also discard the invalid entry and any other entries on the line. To do this, you need to import the exception classes in the `java.util` package, and the best way to do that is to import all of the classes in this package. Now, test this enhancement.
7. Code a static method named `getValidSubtotal` that uses the `hasDouble` method of the `Scanner` class to validate the subtotal entry so the `InputMismatchException` won't occur. This method should require one parameter that receives a `Scanner` object, and it should return a valid subtotal. This method should get an entry from the user, check that it's a valid double value, check that it's greater than zero and less than 10000, display appropriate error messages if it isn't valid, and discard any other user entries whether or not the entry is valid. This should continue until the method gets a valid subtotal entry.

8. After you've written this method, modify the code within the try statement so it uses this method. Then, test this enhancement so you can see that an `InputMismatchException` is no longer caught by the catch block. (When the code in a try block calls a method, any exception that isn't handled by the method is passed back to the code in the try block.)

### **Discard any extra entries for the Continue prompt**

9. Modify the code so the application will work right even if the user enters two or more entries when asked if he wants to continue. To do that, you need to discard any extra entries. Then, test this enhancement.

At this point, the application should be bulletproof. It should only accept valid entries for customer type and subtotal, and it should work even if the user makes two or more entries for a single prompt.

## **Exercise 5-2      Add validation to the Test Score application**

---

In this exercise, you'll add data validation to the Test Score application that you enhanced for chapter 4. To do that, you'll use generic methods that you copy from the Future Value application. This will show you that generic validation methods can be used in a wide range of applications.

1. Open the `EnhancedTestScoreApp` in the `ch04` directory that you developed for exercise 4-3. Then, save the application as `ValidatedTestScoreApp` in the `ch05` directory, change the class name, and compile and run the application to refresh your memory about how it works.
2. Open the `FutureValueValidationApp` in the `ch05` directory. Then, copy the generic `getInt` and `getIntWithinRange` methods from that application and paste them into `ValidatedTestScoreApp`.
3. Use the `getInt` and `getIntWithinRange` methods to validate (1) that the number of test scores that the user enters ranges from 5 through 35, and (2) that each test score ranges from 1 through 100. Then, test this enhancement.
4. Add code that discards any extra entries at the Continue prompt. Then, do your final testing to make sure that the application is bulletproof.



---

## Project 5-2: Calculate the monthly payment on a loan

---

### Console

```
Welcome to the Loan Calculator

DATA ENTRY
Enter loan amount:          ten
Error! Invalid decimal value. Try again.
Enter loan amount:          -1
Error! Number must be greater than 0.0
Enter loan amount:          100000000000
Error! Number must be less than 1000000.0
Enter loan amount:          500000
Enter yearly interest rate: 5.6
Enter number of years:      thirty
Error! Invalid integer value. Try again.
Enter number of years:      -1
Error! Number must be greater than 0
Enter number of years:      100
Error! Number must be less than 100
Enter number of years:      30

FORMATTED RESULTS
Loan amount:                $500,000.00
Yearly interest rate:       5.6%
Number of years:            30
Monthly payment:            $2,870.39

Continue? (y/n):
Error! This entry is required. Try again.
Continue? (y/n): x
Error! Entry must be 'y' or 'n'. Try again.
Continue? (y/n): n

Press any key to continue . . .
```

## Operation

- The Data Entry section prompts the user to enter values for the loan amount, yearly interest rate, and number of years. If the user doesn't enter data that's valid, this section displays an appropriate error message and prompts the user again.
- The Formatted Results section displays a formatted version of the user's entries as well as the formatted result of the calculation.
- The application prompts the user to continue.

## Specifications

- The formula for calculating monthly payment is:

```
double monthlyPayment =
    loanAmount * monthlyInterestRate /
    (1 - 1/Math.pow(1 + monthlyInterestRate, months));
```

- The application should accept decimal entries for the loan amount and interest rate entries.
- The application should only accept integer values for the years field.
- The application should only accept integer and decimal values within the following ranges:

	Greater Than	Less Than
Loan amount:	0	1,000,000
Yearly interest rate:	0	20
Years:	0	100

- The application should only accept a value of "y" or "n" at the Continue prompt.
- If the user enters invalid data, the application should display an appropriate error message and prompt the user again until the user enters valid data.
- The code that's used to validate data should be stored in separate methods. For example:

```
public static double getDoubleWithinRange(Scanner sc, String prompt,
    double min, double max)

public static int getIntWithinRange(Scanner sc, String prompt,
    int min, int max)
```

---

## Project 5-3: Guess a number from 1 to 100

---

### Console

```
Welcome to the Guess the Number Game
+++++

I'm thinking of a number from 1 to 100.
Try to guess it.

Enter number: 50
You got it in 1 tries.
Great work! You are a mathematical wizard.

Try again? (y/n): y

I'm thinking of a number from 1 to 100.
Try to guess it.

Enter number: 50
Way too high! Guess again.

Enter number: 30
Too high! Guess again.

Enter number: 15
Too low! Guess again.

Enter number: 23
Too high! Guess again.

Enter number: 19
Too low! Guess again.

Enter number: 21
Too high! Guess again.

Enter number: 20
You got it in 7 tries.
Not too bad! You've got some potential.

Try again? (y/n):
Error! This entry is required. Try again.
Try again? (y/n): x
Error! Entry must be 'y' or 'n'. Try again.
Try again? (y/n): n

Bye - Come back soon!

Press any key to continue . . .
```



## Operation

- The application prompts the user to enter an int value from 1 to 100 until the user guesses the random number that the application has generated.
- The application displays messages that indicate whether the user's guess is too high or too low.
- When the user guesses the number, the application displays the number of guesses along with a rating. Then, the application asks if the user wants to play again.
- When the user exits the game, the application displays a goodbye message.

## Specifications

- If the user's guess is more than 10 higher than the random number, the application should say, "Way too high!"
- If the user's guess is higher than the random number, the application should say, "Too high!"
- If the user's guess is lower than the random number, the application should say, "Too low!"
- The message that's displayed when the user gets the number should vary depending on the number of guesses. For example:

Number of guesses	Message
=====	=====
<=3	Great work! You are a mathematical wizard.
>3 and <=7	Not too bad! You've got some potential.
>7	What took you so long? Maybe you should take some lessons

- When the user guesses a number, the application should only accept numbers from 1 to 100.
- When the user responds to the Play Again prompt, the application should only accept a value of "y" or "n".
- If the user enters invalid data, the application should display an appropriate error message and prompt the user again until the user enters valid data.
- The code that's used to validate data should be stored in separate methods. For example:
 

```
public static double getDoubleWithinRange(Scanner sc, String prompt,
    double min, double max)

public static int getIntWithinRange(Scanner sc, String prompt,
    int min, int max)
```
- Use the random method of the `java.lang.Math` class to generate a random number.

## Chapter 6

# How to define and use classes

### Objectives

---

#### Applied

- Code the instance variables, constructors, and methods of a class that defines an object.
- Code a class that creates objects from a user-defined class and then uses the methods of the objects to accomplish the required tasks.
- Code a class that contains static fields and methods, and call these fields and methods from other classes.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- Describe the architecture commonly used for object-oriented programs.
- Describe the concept of encapsulation and explain its importance to object-oriented programming.
- Differentiate between an object's identity and its state.
- Describe the basic components of a class.
- Explain what an instance variable is.
- Explain when a default constructor is used and what it does.
- Describe a signature of a constructor or method, and explain what overloading means.
- List four ways you can use the `this` keyword within a class definition.
- Explain the difference between how primitive types and reference types are passed to a method.
- Explain how static fields and static methods differ from instance variables and regular methods.
- Explain what a static initialization block is and when it's executed.

## Summary

---

- In a *three-tiered architecture*, an application is separated into three layers. The *presentation layer* consists of the user interface. The *database layer* consists of the database and the database classes that work with it. And the *middle layer* provides an interface between the presentation layer and the database layer. Its classes are often referred to as *business classes*.
- The *Unified Modeling Language (UML)* is the standard modeling language for working with object-oriented applications. You can use UML *class diagrams* to identify the *fields* and *methods* of a class.
- *Encapsulation* lets you control which fields and methods within a class are *exposed* to other classes. When fields are encapsulated within a class, it's called *data hiding*.
- Multiple *objects* can be created from a single *class*. Each object can be referred to as an *instance* of the class.
- The data that makes up an object can be referred to as its *state*. Each object is a separate entity with its own state.
- A *field* is a variable or constant that's defined at the class level. An *instance variable* is a field that's allocated when an object is instantiated. Each object has a separate copy of each instance variable.
- Every class that contains instance variables has a *constructor* that initializes those variables.
- When you code the methods of a class, you often code public *get* and *set methods* that provide access to the fields of the class.
- If you want to code a method or constructor that accepts arguments, you code a list of *parameters* between the parentheses for the constructor or method. For each parameter, you must include a data type and a name.
- When coding a class, you can use the *this* keyword to refer to the current object.
- When Java passes a *primitive type* to a method, it passes a copy of the value. This is known as *passing by value*. When Java passes an object (a *reference type*) to a method, it passes a reference to the object. This is known as *passing by reference*.
- A *JavaBean* is a special type of Java class that follows a set of coding conventions.
- The name of a method or constructor combined with the list of parameter types is known as the *signature* of the method or constructor. You can *overload* a method or constructor by coding different parameter lists for constructors or methods that have the same name.
- When you use a class that contains only *static fields*, *static methods*, and *static initialization blocks*, you don't create an object from the class. Instead, you call these fields and methods directly from the class.

## Terms

---

multi-layered architecture	state
multi-tiered architecture	access modifier
three-tiered architecture	instance variable
presentation layer	constructor
database layer	get method
middle layer	set method
business rules layer	read-only field
business class	JavaBean
business object	signature
class diagram	overloading
Unified Modeling Language (UML)	default constructor
field	primitive type
method	reference type
encapsulation	pass by value
data hiding	pass by reference
exposing fields and methods	static field
class	static method
object	class field
object diagram	class method
instance of a class	static initialization block
identity	

## Exercise 6-1 Enhance the Line Item application

---

This exercise guides you through the process of testing and enhancing the Line Item application that is presented in this chapter.

1. Open the `LineItemApp`, `Validator`, `Product`, `LineItem`, and `ProductDB` classes that are in the `c:\java1.6\ch06\LineItem` directory and review this code.
2. Compile all five classes. If the compiler throws an error that indicates that it can't "resolve" a class, you need to set your class path correctly as described in chapter 1. Once you've compiled these classes, run the `LineItemApp` class with valid codes like "java", "jps", and "mcb2" to make sure that this application works correctly. Then, test it with an invalid code to see how that works.
3. Add another product to the `ProductDB` class. Its code should be "txtp", its description should be "TextPad", and its price should be \$20.00. Then, compile just this class, and test the `LineItemApp` class again with the new product code. This shows that you can make a change to a class without affecting the other classes that use it.
4. Add a static field to the `LineItem` class that will count the number of objects that are created from the class as shown in figure 6-15. Next, add the code that increments this field each time an object is created from this class, and add a method named `getObjectCount` that gets the value of this field. Then, compile that class.
5. Modify the `LineItemApp` class so it uses the `getObjectCount` method and displays the object count on the console after it displays each line item. Then, compile and run that class to make sure that the changes in steps 4 and 5 work correctly.

## **Exercise 6-2      Use classes that have static methods in the Future Value application**

---

This exercise guides you through the process of modifying the Future Value application so it uses classes that provide static methods.

1. Open the `FutureValueValidationApp` class that's in the `ch05` directory and save it as `FutureValueApp` in the `ch06\FutureValue` directory. Then, change its class name to `FutureValueApp`.
2. Start a new class named `Validator` and save it in the same directory. Move the `getDouble`, `getDoubleWithinRange`, `getInt`, and `getIntWithinRange` methods into the `Validator` class. Next, change the name of the `getDoubleWithinRange` method to `getDouble`, and change the name of the `getIntWithinRange` method to `getInt`. This overloads the `getDouble` and `getInt` methods. Then, compile this class.
3. Modify the `FutureValueApp` class so it uses the methods in the `Validator` class. Then, compile and run that class to make sure that it works correctly.
4. Start a new class named `FinancialCalculations`, and save it in the same directory as the other classes. Move the `calculateFutureValue` method from the `FutureValueApp` class to the `FinancialCalculations` class, and make sure that the method is public. Then, compile this class.
5. Modify the `FutureValueApp` class so it uses the static `calculateFutureValue` method that's stored in the `FinancialCalculations` class. Then, compile and run this class to make sure that the application still works properly.

## **Exercise 6-3      Use objects in the Invoice application**

---

In this exercise, you'll create an `Invoice` class and construct objects from it as you convert the Invoice application to an object-oriented application.

1. Open the `InvoiceApp` and `Validator` classes in the `ch06\Invoice` directory. This is yet another version of the Invoice application. Then, compile and run the classes to see how this application works.
2. Start a new class named `Invoice` and save it in the same directory. Then, write the code for this class so it provides all of the data and all of the processing for an `Invoice` object. Its constructor should require the subtotal and customer type as its only parameters, and it should initialize instance variables for discount percent, discount amount, and invoice total. Its methods should include the required get and set methods, plus a method named `getInvoice` that returns a string that contains all of the data for an invoice in a printable format. When you're done, compile the `Invoice` class.
3. Modify the code in the `InvoiceApp` class so it creates an `Invoice` object and uses its `getInvoice` method to get the formatted data for invoice. That should simplify this class considerably. Then, compile and test this class to make sure that this application works the way it did in step 1.

---

## Project 6-1: Store email addresses and phone numbers

---

### Console

```
Welcome to the Address Book application

1 - List entries
2 - Add entry
3 - Exit

Enter menu number: 1

Name          Email          Phone
-----
Bill Gates    bill@microsoft.com (111) 222-3333
Larry Ellison larry@sun.com    (444) 555-6666
Steve Jobs    steve@apple.com  777-888-9999

1 - List entries
2 - Add entry
3 - Exit

Enter menu number: 2

Enter name: Mike Murach
Enter email address: mike@murach.com
Enter phone number: 800-221-5528

This entry has been saved.

1 - List entries
2 - Add entry
3 - Exit

Enter menu number: 1

Name          Email          Phone
-----
Bill Gates    bill@microsoft.com (111) 222-3333
Larry Ellison larry@sun.com    (444) 555-6666
Steve Jobs    steve@apple.com  777-888-9999
Mike Murach   mike@murach.com  800-221-5528

1 - List entries
2 - Add entry
3 - Exit

Enter menu number: 3

Goodbye.

Press any key to continue . . .
```

## Operation

- If the user selects the first menu option, the application displays the email addresses and phone numbers that have been saved. Then, it displays the menu again.
- If the user selects the second menu option, the application prompts the user to enter a name, email address, and phone number. Then, it displays the menu again.
- If the user selects the third menu option, the application exits.

## Specifications

- Use the `AddressBookIO` class that's provided to get and save entries. This class contains two static methods that you can use to read and write data to the `address_book.txt` file. They are:

```
// get a String that displays all entries in columns
public static String getEntriesString()

// save an AddressBookEntry object to the file
public static boolean saveEntry(AddressBookEntry entry)
```

- If necessary, you can open the `address_book.txt` file in a text editor to debug this application.
- Create a class named `AddressBookEntry` to store the data for each entry. This class should contain these instance variables:

```
private String name;
private String emailAddress;
private String phoneNumber;
```
- And it should contain get and set methods for each of these instance variables.
- Create a class named `AddressBookEntryApp`. This class should display the menu and respond to the user's menu choices using the `AddressBookEntry` and `AddressBookIO` classes as necessary.
- Create a class named `Validator` that contains static methods that can be used to validate the data in this application. The user should only be able to select one of the menu choices, and the user must enter some text for name, email address, and phone number.

---

## Project 6-2: Calculate a circle's circumference and area

---

### Console

```
Welcome to the Circle Tester

Enter radius: 3
Circumference: 18.85
Area:          28.27

Continue? (y/n): y

Enter radius: 6
Circumference: 37.7
Area:          113.1

Continue? (y/n): n

Goodbye. You created 2 Circle object(s).

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter the radius of a circle.
- If the user enters invalid data, the application displays an appropriate error message and prompts the user again until the user enters valid data.
- When the user enters a valid radius, the application calculates and displays the circumference and area of the circle to the nearest 2 decimal places.
- The application prompts the user to continue.
- When the user chooses not to continue, the application displays a goodbye message that indicates the number of Circle objects that were created by the application.

### Specifications

- Create a class named Circle to store the data about this circle. This class should contain these constructors and methods:

```
public Circle(double radius)
public double getCircumference()
public String getFormattedCircumference()
public double getArea()
public String getFormattedArea()
private String formatNumber(double x)
public static int getObjectCount()
```

- The formulas for calculating circumference and area are:

```
circumference = 2 * pi * radius
area = pi * radius2
```

- For the value of pi, use the PI constant of the java.lang.Math class.
- Create a class named CircleApp that gets the user input, creates a Circle object, and displays the circumference and area.
- Create a class named Validator like the one shown in chapter 6 and use its static methods to validate the data in this application.



---

## Project 6-3: Roll the dice

---

### Console

```
Welcome to the Paradise Roller application

Roll the dice? (y/n): y

Roll 1:
2
5
Craps!

Roll again? (y/n): y

Roll 2:
2
1

Roll again? (y/n): y

Roll 3:
4
6

Roll again? (y/n): y

Roll 4:
6
6
Box cars!

Roll again? (y/n): y

Roll 5:
1
1
Snake eyes!

Roll again? (y/n): n

Press any key to continue . . .
```

### Operation

- If the user chooses to roll the dice, the application rolls two six-sided dice, displays the results of each, and asks if the user wants to roll again.

## Specifications

- Create a class named `Die` to store the data about each die. This class should contain these constructors and methods:

```
public Die()                // default to a six-sided die
public Die(int sides)       // allow a variable number of sides
public void roll()
public int getValue()
```

- Create a class named `PairOfDice` to store two dice. This class should contain two instance variables of the `Die` type, an instance variable that holds the sum of the two dice, and these constructors and methods:

```
public PairOfDice()         // default to six-sided dice
public PairOfDice(int sides) // allow a variable number of sides
public void roll()
public int getValue1()      // get value of die1
public int getValue2()      // get value of die2
public int getSum()         // get the sum of both dice
```

- You can use the `random` method of the `Math` class to generate a random number from 1 to the number of sides on a die like this:

```
int value = (int) (Math.random() * sides);
```

- Create a class named `DiceRollerApp` that uses the `PairOfDice` class to roll the dice. This class should display special messages for craps (sum of both dice is 7), snake eyes (double 1's), and box cars (double 6's). For this application, assume that two six-sided dice are used.
- Create a class named `Validator` that contains static methods that can be used to validate the data in this application.

## Chapter 7

# How to work with inheritance

### Objectives

---

#### Applied

- Given the specifications for an application that uses inheritance, create the required classes.
- Given the specifications for an abstract class or method, write the code for the class and the class that inherits it.
- Given the specifications for a final class, method, or parameter, write the code for the class and the class that inherits it.
- Write the code necessary for a class to override the equals method of the Object class so you can test two objects created from that class for equality based on the data they contain.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- In general, explain how inheritance works.
- Explain what it means for a subclass to extend a superclass.
- Explain how inheritance is used within the Java API classes.
- Explain why methods such as toString and equals are available to all objects and when you might override these methods.
- Describe two ways that you can use inheritance in your applications.
- Describe the accessibility that's provided by the access modifiers you can use for the members of a class.
- Explain what polymorphism is and how it works.
- Describe two ways that you can get information about an object's type.
- Explain when it's necessary to use explicit casting when working with objects created from derived classes.
- Explain how abstract classes and methods work.
- Explain how final classes, methods, and parameters work.

## Summary

---

- *Inheritance* lets you create a new class based on an existing class. The existing class is called the *superclass*, *base class*, or *parent class*, and the new class is called the *subclass*, *derived class*, or *child class*.
- A subclass inherits all of the fields and methods of its superclass. The subclass can *extend* the superclass by adding its own fields and methods, and it can *override* a method with a new version of the method.
- All classes inherit the `java.lang.Object` class, which provides methods such as `toString`, `equals`, and `getClass`.
- You can use *access modifiers* to limit the accessibility of the fields and methods declared by a class. *Protected members* can be accessed only by classes in the same package or by subclasses.
- In a subclass, you can use the `super` keyword to access the fields, constructors, and methods of the superclass.
- *Polymorphism* is a feature of inheritance that lets you treat subclasses as though they were their superclass.
- You can call the `getClass` method from any object to get a `Class` object that contains information about that object.
- You can use the `instanceof` operator to check if an object is an instance of a particular class.
- Java can implicitly cast a subclass type to its superclass type, but you must use explicit casting to cast a superclass type to a subclass type.
- *Abstract classes* provide code that can be used by subclasses. In addition, they can specify *abstract methods* that must be implemented by subclasses.
- You can use the `final` keyword to declare *final classes*, *final methods*, and *final parameters*. No class can inherit a final class, no method can override a final method, and no statement can assign a new value to a final parameter.

## Terms

---

inheritance	hash code
subclass	garbage collector
derived class	access modifier
child class	protected member
superclass	polymorphism
parent class	late binding
base class	runtime type identification (RTTI)
extending a superclass	abstract class
overriding a method	abstract method
frame	final class
inheritance hierarchy	final method
Abstract Window Toolkit (AWT)	final parameter
Swing	

## Exercise 7-1      Look at a class that inherits the JFrame class

---

This exercise lets you view and run a class that inherits the `javax.swing.JFrame` class.

1. Open the `ProductFrame` class that's stored in the `c:\java1.6\ch07\FrameTest` directory. When you review this code, notice how it inherits the `JFrame` class and calls methods inherited from this class. In addition, note that this class contains a `main` method that creates an instance of the `ProductFrame` and displays that instance.
2. Compile and run this class. This should display a frame. When you click on its close button, the frame should close. In section 4, you'll learn more about working with frames like this one. In particular, you'll learn how to add components such as buttons, labels, and text boxes.
3. Make a list of all the methods of the `JFrame` class that are called in this application. Then, using the documentation for the `JFrame` class from the Java API documentation, indicate whether the method is declared by the `JFrame` class or inherited from one of its superclasses. If the method is inherited, indicate which class it is inherited from.
4. Use your research from step 3 to determine which class inherited by the `JFrame` class actually declares the `setVisible` method. Then, modify the code in the `main` method so that the frame variable is declared as that type rather than as a `JFrame` type. Compile and run the application to verify that it still works properly.

## Exercise 7-2      Create a Product application that uses inheritance

---

In this exercise, you'll create a `Product` application like the one presented in this chapter that uses inheritance. However, you will add an additional kind of product: compact discs. To make this application easier to develop, we'll give you most of the starting classes.

### Create a new subclass named `CompactDisc`

1. Open the classes in the `c:\java1.6\ch07\Product` directory and review the code.
2. Add a class named `CompactDisc` that inherits the `Product` class. This new class should work like the `Book` and `Software` classes, but it should include public `get` and `set` methods for a private instance variable named `artist`. In addition, it should include a `toString` method that overrides the `toString` method in the `Product` class. This method should append the artist name to the end of the string.
3. Compile the `CompactDisc` class to make sure that it doesn't contain any syntax errors.

### Modify the ProductDB class so it returns a CompactDisc object

4. Modify the ProductDB class so it creates at least one CompactDisc object. For example, this object might contain the following information:

```
Code:          sgtp
Description:   Sgt. Pepper's Lonely Hearts Club Band
Price:        $15.00
Artist:       The Beatles
```

5. Compile this class and run the application to make sure it works.

### Add a protected variable

6. Open the Product class and change the access modifier for the count variable from public to protected.
7. Compile this class, and then run the application one more time to make sure that the count is maintained properly.

## Exercise 7-3      Modify the Product class to use the abstract keyword

---

In this exercise, you'll change the Product class in the Product application to an abstract class to see how that works. Then, you'll add an abstract method and implement it in the Book, Software, and CompactDisc subclasses.

### Change the Product class to an abstract class

1. Still working with the Product application of exercise 7-2, add the abstract keyword to the Product class declaration and compile the class.
2. Open the ProductApp class, and add this statement before the statement that calls the getProduct method:  

```
Product pTest = new Product();
```
3. Compile the ProductApp class. You should get a message that says that the Product class is declared as abstract and cannot be instantiated.
4. Delete the statement you just added and compile the class again. Then, run the application to make sure it works.

### Add an abstract method to the Product class

5. Add an abstract method named getDisplayText to the Product class. This method should accept no parameters, and it should return a String object. Compile this class.
6. Rename the toString methods in the Book, Software, and CompactDisc classes to getDisplayText, and compile these classes.
7. Modify the ProductApp class so it calls the getDisplayText method of a product object instead of the toString method. Then, compile this class, and run the application to be sure it works correctly.

## Exercise 7-4      **Modify the Book class to use the final keyword**

---

In this exercise, you'll change the Book class in the Product application to a final class to see that a final class can't be inherited. Then, you'll create a final method to see that it can't be overridden.

### **Change the Book class to a final class**

1. Still working with the Product application, add the final keyword to the Book class declaration and compile this class.
2. Create a class named UsedBook that inherits the Book class. You don't need to include any code in the body of this class. Then, compile this class. When you do, you should get a message that says the Book class can't be inherited because that class is final.

### **Add a final method**

3. Remove the final keyword from the Book class declaration. Then, add the final keyword to the getDisplayText method of the Book class and compile this class.
4. Add a getDisplayText method to the UsedBook class to override the getDisplayText method of the Book class. You don't need to include any code in the body of this method. Then, compile this class. When you do, you should get a message that says the getDisplayText method can't be overridden because that method is final.

## Exercise 7-5      **Code an equals method for the Product and LineItem classes**

---

In this exercise, you'll add an equals method to the Product and LineItem classes that you can use to compare the instance variables of two objects.

1. Open the EqualsTestApp class in the c:\java1.6\ch07\EqualsTest directory. This application creates and compares two Product objects and two LineItem objects using the equals method. Review this code to see how it works.
2. Compile the EqualsTestApp class, and run the application. Since the equals method isn't overridden in the Product or LineItem class, the output from this application should indicate that the comparisons are based on object references and not the data the objects contain.
3. Open the Product class, and add an equals method like the one shown in figure 7-15. Then, compile the Product class, and run the EqualsTestApp class again. This time, the output should indicate that the products are being compared based on their data and not their references.
4. Repeat step 2 for the LineItem class. This time, the comparisons for both the products and line items should be based on their data.

---

## Project 7-1: Create an object-oriented validation class

---

### Console

```
Welcome to the Validation Tester application

Int Test
Enter an integer between -100 and 100: x
Error! Invalid integer value. Try again.
Enter an integer between -100 and 100: -101
Error! Number must be greater than -101
Enter an integer between -100 and 100: 101
Error! Number must be less than 101
Enter an integer between -100 and 100: 100

Double Test
Enter any number between -100 and 100: x
Error! Invalid decimal value. Try again.
Enter any number between -100 and 100: -101
Error! Number must be greater than -101.0
Enter any number between -100 and 100: 101
Error! Number must be less than 101.0
Enter any number between -100 and 100: 100

Required String Test
Enter your email address:
Error! This entry is required. Try again.
Enter your email address: joelmurach@yahoo.com

String Choice Test
Select one (x/y):
Error! This entry is required. Try again.
Select one (x/y): q
Error! Entry must be 'x' or 'y'. Try again.
Select one (x/y): x

Press any key to continue . . .
```

### Operation

- This application prompts the user to enter a valid integer within a specified range, a valid double within a specified range, a required string, and one of two strings. If an entry isn't valid, the application displays an appropriate error message.



## Specifications

- Create a class named `OOValidator` that contains these constructors and methods:

```
public OOValidator(Scanner sc)
public int getInt(String prompt)
public int getIntWithinRange(String prompt, int min, int max)
public double getDouble(String prompt)
public double getDoubleWithinRange(String prompt,
    double min, double max)
```

You can use the `Validator` class that's provided as a starting point for coding the methods in this class.

- Create a class named `MyValidator` that extends the `OOValidator` class. This class should add two new methods:

```
public String getRequiredString(String prompt)
public String getChoiceString(String prompt,
    String s1, String s2)
```

- Create a class named `ValidatorTestApp` and use it to test the methods in the `Validator`, `OOValidator`, and `MyValidator` classes.

---

## Project 7-2: Work with customer and employee data

---

### Console

```
Welcome to the Person Tester application

Create customer or employee? (c/e): c

Enter first name: Frank
Enter last name: Jones
Enter email address: frank44@hotmail.com
Customer number: M10293

You entered:
Name: Frank Jones
Email: frank44@hotmail.com
Customer number: M10293

Continue? (y/n): y

Create customer or employee? (c/e): e

Enter first name: Anne
Enter last name: Prince
Enter email address: anne@murach.com
Social security number: 111-11-1111

You entered:
Name: Anne Prince
Email: anne@murach.com
Social security number: 111-11-1111

Continue? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the user to enter a customer or an employee.
- If the user selects customer, the application asks for name, email, and customer number.
- If the user selects employee, the application asks for name, email, and social security number.
- When the user finishes entering data for a customer or employee, the application displays the data that the user entered.

## Specifications

- Create an abstract `Person` class that stores first name, last name, and email address. This class should provide a no-argument constructor, get and set methods for each piece of data, and it should override the `toString` method so it returns the first name, last name, and email fields in this format:  
**Name: Frank Jones**  
**Email: frank44@hotmail.com**
- In addition, it should contain an abstract method named `getDisplayText` that returns a string.
- Create a class named `Customer` that inherits the `Person` class. This class should store a customer number, it should provide get and set methods for the customer number, it should provide a no-argument constructor, and it should provide an implementation of the `getDisplayText` method. The `getDisplayText` method should return a string that consists of the string returned by the `toString` method of the `Person` class appended with the `Customer` number like this:  
**Name: Frank Jones**  
**Email: frank44@hotmail.com**  
**Customer number: M10293**
- Create a class named `Employee` that inherits the `Person` class. This class should store a social security number, it should provide get and set methods for the social security number, it should provide a no-argument constructor, and it should provide an implementation of the `getDisplayText` method. The `getDisplayText` method should return a string that consists of the string returned by the `toString` method of the `Person` class appended with the `Employee`'s social security number like this:  
**Name: Anne Prince**  
**Email: anne@murach.com**  
**Social security number: 111-11-1111**
- Create a class named `PersonApp` that prompts the user as shown in the console output. This class should create the necessary `Customer` and `Employee` objects from the data entered by the user, and it should use these objects to display the data to the user. To print the data for an object to the console, this application should use a static method named `print` that accepts a `Person` object.
- Use the `Validator` class or a variation of it to validate the user's entries.

## Chapter 8

# How to work with interfaces

### Objectives

---

#### Applied

- Create and implement an interface and use the resulting class in an application.
- Create a class that inherits another class and implements one or more interfaces.
- Create an interface that inherits other interfaces.
- Implement the Cloneable interface for any classes that you've created and then use those classes in an application.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- In general, explain how interfaces work.
- Describe two advantages of using an interface over using an abstract class.
- Name two general-purpose interfaces defined by the Java API and explain what they do.
- Explain what a tagging interface is and how it's used.
- Describe two ways that you can use an interface as a parameter.
- Explain how you can use interfaces and the factory pattern to separate the presentation layer of an application from the database layer.
- Explain the difference between a mutable object and an immutable object.

### Summary

---

- An *interface* is a special type of coding element that can contain static constants and abstract methods. Although a class can only inherit one other class, it can *implement* more than one interface.
- To implement an interface, a class must implement all the abstract methods defined by the interface. An interface can also inherit other interfaces, in which case the implementing class must also implement all the methods of the inherited interfaces.
- An interface defines a Java type. Because of that, you can use an object that's created from a class that implements an interface anywhere that interface is expected.
- When you *clone* an object, you make an identical copy of the object.
- Before you can use the clone method of the Object class, you need to implement the Cloneable interface. Then, you can override the clone method so it is public and so it works correctly with *mutable* objects.

## Terms

---

multiple inheritance  
interface  
implement an interface  
tagging interface  
factory pattern  
clone an object  
immutable object  
mutable object

## Exercise 8-1 Create and work with interfaces

---

In this exercise, you'll create and implement the `DepartmentConstants` interface presented in this chapter. You'll also create and implement an interface named `Displayable` that's similar to the `Printable` interface.

### Create the interfaces

1. Open the classes in the `c:\java1.6\ch08\DisplayableTest` directory.
2. Add an interface named `DepartmentConstants` that contains the three constants shown in figure 8-4. Compile the interface.
3. Add an interface named `Displayable`. This interface should contain a single method named `getDisplayText` that returns a `String`. Compile this interface.

### Implement the interfaces

4. Edit the `Product` class so it implements the `Displayable` interface. The `getDisplayText` method in this class should format a string that can be used to display the product information. When you're done, compile this class.
5. Edit the `Employee` class so it implements the `DepartmentConstants` and `Displayable` interfaces. The `getDisplayText` method in this class should work like the one in the `Product` class, and it should use the constants in the `DepartmentConstants` interface to include the department name in the return value. When you're done, compile this class.

### Use the classes that implement the interfaces

6. Display the `DisplayableTestApp` class.
7. Add code to this class that creates an `Employee` object, assigns it to a `Displayable` variable, and displays the information in the `Employee` object at the console. To get the information for an employee, you'll need to use the `getDisplayText` method of the `Displayable` interface.
8. Compile and run the application to make sure that it displays the employee information.
9. Repeat steps 7 and 8 for a `Product` object.

## Exercise 8-2 Use an interface as a parameter

---

In this exercise, you'll use the `Displayable` interface you created and implemented in exercise 8-1 as a method parameter.

1. Open the classes in the `c:\java 1.6\ch08\DisplayableTest` directory and display the `DisplayableTestApp` class.
2. Add a method named `displayMultiple` that accepts a `Displayable` object and an integer and returns a string. The string returned by this method should contain the number of occurrences specified by the `int` parameter of the object specified by the `Displayable` parameter.

3. Modify the code in the main method so that it uses the `displayMultiple` method to display one occurrence of the employee information and two occurrences of the product information.
4. Compile and run the application to make sure it works correctly.

### **Exercise 8-3      Add an update function to the Product Maintenance application**

---

In this exercise, you'll review the Product Maintenance application presented in this chapter. Then, you'll add an update function to this application.

#### **Review and run the application**

1. Open the files in the `c:\java1.6\ch08\ProductMaintenance` directory.
2. Review the code in each file to see how it works.
3. Run the application and try each of its functions. When you're comfortable with how it works, exit from the application.

#### **Modify the application so it includes an update function**

4. Add code to the `ProductMaintApp` class that lets the user update an existing product. To do that, you'll need to add an update command to the list of commands, and you'll need to add an `updateProduct` method that's executed if the user enters this command.
5. The `updateProduct` method should start by getting a valid product code from the user. Then, it should let the user update either the product's description or price. After the description or price of the `Product` object is updated, the `updateProduct` method should call the `updateProduct` method of the `ProductDAO` object to update the product.
6. Compile and run the `ProductMaintApp` class to make sure it works correctly.

### **Exercise 8-4      Implement the Cloneable interface**

---

In this exercise, you'll implement the `Cloneable` interface for the `Product` and `LineItem` classes.

1. Open the classes in the `c:\java1.6\ch08\CloneableTest` directory. Display the `ProductCloneApp` class and review its code. Then, try to compile this class. The compilation will fail because the `Cloneable` interface hasn't been implemented in the `Product` class.
2. Implement the `Cloneable` interface for the `Product` class. Then, compile the `Product` and `ProductCloneApp` classes. If you implemented the `Cloneable` interface correctly, the `ProductCloneApp` class should compile.
3. Run the `ProductCloneApp` class to make sure it works correctly.
4. Repeat steps 1 through 3 for the `LineItemCloneApp` and `LineItem` classes.

---

## Project 8-1: Count alligators and clone sheep

---

### Console

```
Counting alligators...

1 alligator
2 alligator
3 alligator

Counting sheep...

1 Blackie
2 Blackie

1 Dolly
2 Dolly
3 Dolly

1 Blackie

Press any key to continue . . .
```

### Operation

- This application uses an Alligator class that implements a Countable interface to display Alligator objects as shown above.
- This application uses a Sheep class that implements a Countable interface and the Cloneable interface to display and clone Sheep objects as shown above.

### Specifications

- Create an interface named Countable that can be used to count an object. This interface should include these methods:

```
void incrementCount()
void resetCount()
int getCount()
String getCountString()
```
- Create a class named Alligator that implements the Countable interface. This class should include an instance variable that stores the count and a method that returns the formatted count.
- Create a class named CountUtil. This class should include a static method that lets you count any Countable objects a specified number of times. For example:

```
public static void count(Countable c, int maxCount)
```
- Create a class named CountTestApp that uses the CountUtil class to count an Alligator object 3 times as shown above.
- Create a class named Sheep that implements the Countable and Cloneable interfaces. This class should include an instance variable that stores the count and the name of the sheep, and it should provide methods that can set and get the name of the sheep.
- Modify the CountTestApp class so it (a) counts the first sheep 2 times, (b) clones the first sheep, changes the name, and counts it 3 times, and (c) counts the first sheep again 1 time.



---

## Project 8-2: Calculate the monthly balances of two bank accounts

---

### Console

```
Welcome to the Account application

Starting Balances
Checking: $1,000.00
Savings:  $1,000.00

Enter the transactions for the month

Withdrawal or deposit? (w/d): w
Checking or savings? (c/s): c
Amount?: 500

Continue? (y/n): y

Withdrawal or deposit? (w/d): d
Checking or savings? (c/s): s
Amount?: 200

Continue? (y/n): n

Monthly Payments and Fees
Checking fee:                $1.00
Savings interest payment:   $12.00

Final Balances
Checking: $499.00
Savings:  $1,212.00

Press any key to continue . . .
```

### Operation

- The application begins by displaying the starting balances for a checking and savings account.
- The application prompts the user to enter the information for a transaction, including whether a withdrawal or deposit is to be made, whether the transaction will be posted to the checking or savings account, and the amount of the transaction.
- When the user finishes entering deposits and withdrawals, the application displays the fees and payments for the month followed by the final balances for the month.

## Specifications

- Create interfaces named `Depositable`, `Withdrawable`, and `Balanceable` that specify the methods that can be used to work with accounts. The `Depositable` interface should include this method:

```
public void deposit(double amount)
```

The `Withdrawable` interface should include this method:

```
public void withdraw(double amount)
```

And the `Depositable` interface should include these methods:

```
public double getBalance()
public void setBalance(double amount)
```

- Create a class named `Account` that implements all three of these interfaces. This class should include an instance variable for the balance.
- Create a class named `CheckingAccount` that inherits the `Account` class. This class should include an instance variable for the monthly fee that's initialized to the value that's passed to the constructor. This class should also include methods that subtract the monthly fee from the account balance and return the monthly fee.
- Create a class named `SavingsAccount` that inherits the `Account` class. This class should include instance variables for the monthly interest rate and the monthly interest payment. The monthly interest rate should be initialized to the value that's passed to the constructor. The monthly interest payment should be calculated by a method that applies the payment to the account balance. This class should also include a method that returns the monthly interest payment.
- Create a class named `Transactions` that contains two static methods for depositing and withdrawing funds from either type of account:

```
public class Transactions
{
    public static void deposit(Depositable account,
        double amount)
    {
        account.deposit(amount);
    }

    public static void withdraw(Withdrawable account,
        double amount)
    {
        account.withdraw(amount);
    }
}
```

- Create a class named `AccountApp` that prompts the user for a transaction, posts the transaction, and displays the information shown in the console output. Create the necessary objects for each transaction, and post the transaction using the appropriate method of the `Transactions` class.
- Use the `Validator` class or a variation of it to validate the user's entries. This validation code should not allow the user to withdraw more than the current account balance.

---

## Project 8-3: Reduce the linkage to the presentation layer

---

### Console

```
Welcome to the Console Tester application

Int Test
Enter an integer between -100 and 100: 50

Double Test
Enter any number between -100 and 100: 50

Required String Test
Enter your email address: joel@murach.com

String Choice Test
Select one (x/y): x

Press any key to continue . . .
```

### Operation

- This application prompts the user to enter a valid integer within a specified range, a valid double within a specified range, a required string, and one of two strings. If a user entry isn't valid, the application displays an appropriate error message.

### Specifications

- Create an interface named `ConsoleOutput` that specifies the methods that can be used to get input from the user as follows:

```
void print(String s)
void println(String s)
void println()
```

- Create an interface named `ConsoleInput` that specifies the methods that can be used to get input from the user as follows:

```
int getInt(String prompt);
int getIntWithinRange(String prompt, int min, int max);

double getDouble(String prompt);
double getDoubleWithinRange(String prompt, double min, double max);

String getRequiredString(String prompt);
String getChoiceString(String prompt, String s1, String s2);
```

- Create an interface named `ConsoleIO` that inherits the `ConsoleInput` and `ConsoleOutput` classes.
- Create a class named `MyConsole` that implements the `ConsoleIO` interface. The constructor for this class should create an instance of the `Scanner` class that gets input from the standard input stream. The methods for this class should let you get valid input from the user and display output to the user.
- Code an `IOFactory` class that contains a method named `getConsoleIO` that returns an instance of a class that implements the `ConsoleIO` interface. For this project, return an instance of the `MyConsole` class.

- Create a `ConsoleTestApp` class that prompts the user as shown in the console output. This class should use the `IOFactory` class to get a `ConsoleIO` object. Then, this class should use the methods of the `ConsoleIO` object to print output to the console and to get input from the console. The code in this class shouldn't use the `Scanner` class or the `System.out` object directly.

### Enhancements

- Add methods to the `ConsoleInput` and `ConsoleOutput` interfaces so that they provide more input and output possibilities. For example, you could include a method in the `ConsoleInput` interface that accepts a prompt and three string values. Then, implement these methods in any classes that implement the `ConsoleIO` interface.
- Code a class named `JConsole` that implements the `ConsoleIO` interface by using the `showInputDialog` method of the `JOptionPane` class. (You'll have to research this method on your own.) Then, edit and recompile the `IOFactory` class so that the `ConsoleTestApp` uses the new class.

## Chapter 9

# Other object-oriented programming skills

### Objectives

---

#### Applied

- Add two or more classes to a package and make the classes in that package available to other classes.
- Add javadoc comments to the classes in one or more packages and generate the documentation for those packages.
- Use your web browser to view the documentation you added to a package.
- Code more than one class per file. When necessary, use nested classes.
- Declare and use an enumeration.
- Enhance an enumeration by adding methods that override the methods of the Java and Enum classes. Use methods of the enumeration constants when necessary.
- Use a static import to import the constants of an enumeration or the static fields and methods of a class.

#### Knowledge

- List three reasons that you might store classes in a package.
- Describe the general procedure for creating a directory structure for a package.
- Explain why you might add javadoc comments to the packages you create.
- Explain the purpose of using HTML and javadoc tags within a javadoc comment.
- Explain when you might code two or more classes in the same file and describe the advantage and disadvantage of doing that.
- Describe the difference between an inner class and a static inner class in terms of how they're related to the outer class.
- Explain what a local class is.
- Explain what an enumeration is and how you use one.
- Explain what static imports are and how you use them.

## Summary

---

- You can organize the classes in your application by using a package statement to add them to a *package*. Then, you can use import statements to make the classes in that package available to other classes.
- You can use *javadoc comments* to document a class and its fields, constructors, and methods. Then, you can use the javadoc command to generate HTML-based documentation for your class.
- When two or more classes are closely related, it sometimes makes sense to store them all in one file or to *nest* them.
- You can use an *enumeration* to define a set of related constants as a type. Then, you can use the constants in the enumeration anywhere the enumeration is allowed.
- You can use *static imports* to import the constants of an enumeration or the static fields and methods of a class. Then, you can refer to the constants, fields, and methods without qualification.

## Terms

---

Java Archive (JAR) file	static inner class
javadoc comment	member class
HTML tag	local class
javadoc tag	enumeration
nested classes	type-safe
outer class	static import
inner class	

## Exercise 9-1      Package the classes in an application

---

This exercise guides you through the process of organizing the Product, LineItem, ProductDB, and Validator classes into packages. Then, it shows you how to modify the LineItemApp class so it uses those packages.

1. Copy the java files for the LineItemApp, LineItem, Product, ProductDB, and Validator classes from the c:\java1.6\ch06\LineItem directory to the c:\java1.6\ch09\LineItem directory.
2. Create subdirectories named murach\business, murach\database, and murach\presentation. Then, move the LineItem, Product, ProductDB, and Validator files into their appropriate subdirectories as shown in figure 9-1.
3. Add package and import statements to the Product, LineItem, ProductDB, and Validator classes as shown in figure 9-1.
4. Use the command prompt to compile these classes as shown in figure 9-2. Be sure to compile the Product class before you compile the ProductDB class. When you compile the LineItem class, it should compile both the Product and LineItem classes.
5. Open the LineItemApp class that's stored in the c:\java1.6\ch09\LineItem directory, and add import statements to import the three packages you just created. Then, compile and run this class to make sure the application is working correctly. When you're sure it is, close the class.

6. Open the `LineItemApp` class that's stored in the `c:\java1.6\ch09\LineItemTester` directory, and try to compile this class. You should get several compile-time errors. That's because the `LineItemApp` class can't find the three packages even though it includes the proper import statements.
7. Use a command window to create a JAR file named `murach.jar` that contains all three packages as shown in figure 9-3. Then, move the `murach.jar` file to the `\jre\lib\ext` subdirectory of your SDK directory.
8. Try to compile the `LineItemApp` class that's stored in the `LineItemTester` directory again. This time, it should compile. Then, run this class to make sure it works correctly.

## Exercise 9-2 Document the murach packages

---

This exercise guides you through the process of adding javadoc comments to the `Product` and `LineItem` classes and using the javadoc tool to generate the API documentation for all the murach packages.

1. Open the `Product` and `LineItem` classes that are stored in the `c:\java1.6\ch09\LineItem\murach\business` directory.
2. Add javadoc comments like the ones shown in figure 9-4 for the `LineItem` class and its constructors and methods. Then, compile the class.
3. Add javadoc comments like the ones shown in figure 9-5 for the `Product` class and its constructor and methods. Then, compile the class.
4. Open a command window and use the `javadoc` command to generate the documentation for the `murach.business`, `murach.database`, and `murach.presentation` classes as shown in figure 9-6. When you're done, this documentation should be stored in the `c:\java1.6\ch09\LineItem\docs` directory.
5. Start your web browser, navigate to the `c:\java1.6\ch09\LineItem\docs` directory, and open the `index.html` page.
6. Click on the `Validator` class in the lower left frame to see the documentation that's generated for a class by default.
7. Click on the `murach.business` package to display just the `LineItem` and `Product` classes in the lower left frame. Then, click on the `LineItem` class to display its documentation. Notice that the details for the methods don't include a description of the parameters or return values.
8. Click on the `Product` class to display its documentation. Scroll to the method details to see how the descriptions of the parameters and return values are displayed.
9. When you're done experimenting, close your web browser.

### **Exercise 9-3      Code more than one file per class**

---

In this exercise, you'll combine the code for two classes into a single file.

1. Open the code for the Customer and Address classes that are stored in the `c:\java1.6\ch09\Classes` directory. Cut the code from the Address class and paste it at the end of the Customer class. Delete the public modifier from the declaration of the Customer class, and save the file.
2. Delete the Address.java file. At this point, the `c:\java1.6\ch09\Classes` directory should only contain the Customer.java file and no .class files.
3. Compile the Customer class. Then, view the files in the `c:\java1.6\ch09\Classes` directory. Now, there should be .class files for both the Customer and Address classes. This shows that the Customer.java file now stores two classes.

### **Exercise 9-4      Create and use an enumeration**

---

In this exercise, you'll create an enumeration and then use it in a test application.

1. Create an enumeration named CustomerType, and save it in the `c:\java1.6\ch09\Enumeration` directory. This enumeration should contain constants that represent three types of customers: retail, trade, and college.
2. Open the CustomerTypeApp class in the `c:\java 1.6\ch09\Enumeration` directory. Then, add a method to this class that returns a discount percent (.10 for retail, .30 for trade, and .20 for college) depending on the CustomerType variable that's passed to it.
3. Add code to the main method that declares a CustomerType variable, assigns a customer type to it, gets the discount percent for that customer type, and displays the discount percent. Compile and run the application to be sure that it works correctly.
4. Add a statement to the main method that displays the string returned by the toString method of the customer type. Then, compile and run the application again to see the result of this method.
5. Add a toString method to the CustomerType enumeration. This method should return a string that contains "Retail customer," "Trade customer," or "College customer" depending on the customer type. Compile this class, then run the CustomerTypeApp class again to view the results of the toString method.



---

## Project 9-1: Create, package, and document the Console class

---

### Console

```
Welcome to the Console Tester application

Int Test
Enter an integer between -100 and 100:
Error! This entry is required. Try again.
Enter an integer between -100 and 100: x
Error! Invalid integer value. Try again.
Enter an integer between -100 and 100: -101
Error! Number must be greater than -101
Enter an integer between -100 and 100: 101
Error! Number must be less than 101
Enter an integer between -100 and 100: 50

Double Test
Enter any number between -100 and 100:
Error! This entry is required. Try again.
Enter any number between -100 and 100: x
Error! Invalid decimal value. Try again.
Enter any number between -100 and 100: -101
Error! Number must be greater than -101.0
Enter any number between -100 and 100: 101
Error! Number must be less than 101.0
Enter any number between -100 and 100: 50

Required String Test
Enter your email address:
Error! This entry is required. Try again.
Enter your email address: joelmurach@yahoo.com

String Choice Test
Select one (x/y):
Error! This entry is required. Try again.
Select one (x/y): q
Error! Entry must be 'x' or 'y'. Try again.
Select one (x/y): x

Press any key to continue . . .
```

### Operation

- This application prompts the user to enter a valid integer within a specified range, a valid double within a specified range, a required string, and one of two strings. If a user entry isn't valid, the application displays an appropriate error message.

## Specifications

- Create a class named `Console` that can be used to display output to the user and get input from the user. Feel free to reuse your best code from any previous exercises or projects. At a minimum, this class should include these methods:

```
// for output
public void print(String s);
public void println(String s);
public void println();

// for input
public String getRequiredString(String prompt);
public String getChoiceString(String prompt, String s1, String s2);
public int getInt(String prompt);
public int getIntWithinRange(String prompt, int min, int max);
public double getDouble(String prompt);
public double getDoubleWithinRange(String prompt, double min, double
max);
```

- Create a class named `ConsoleTestApp` that tests the `Console` application to make sure it's working correctly as shown in the console output. Feel free to reuse your best code from any previous exercises or projects.
- Store the `Console` class in a package named `yourLastName.util`
- Then, add an import statement for this package to the `ConsoleTestApp` class.
- Add javadoc comments to the `Console` class. These comments should document the purpose, author, and version of the class. It should also document the function of each method, including any parameters accepted by the method and any value it returns.
- Generate the documentation for the `Console` class and store it in a directory named `docs` that is a subdirectory of the root directory for this project.

---

## Project 9-2: Create the Roshambo game

---

### Console

```
Welcome to the game of Roshambo

Enter your name: Joel

Would you like to play Bart or Lisa? (B/L): b

Rock, paper, or scissors? (R/P/S): r

Joel: rock
Bart: rock
Draw!

Play again? (y/n): y

Rock, paper, or scissors? (R/P/S): p

Joel: paper
Bart: rock
Joel wins!

Play again? (y/n): y

Rock, paper, or scissors? (R/P/S): s

Joel: scissors
Bart: rock
Bart wins!

Play again? (y/n): n

Press any key to continue . . .
```

### Operation

- The application prompts the player to enter a name and select an opponent.
- The application prompts the player to select rock, paper, or scissors. Then, the application displays the player's choice, the opponent's choice, and the result of the match.
- The application continues until the user doesn't want to play anymore.
- If the user makes an invalid selection, the application should display an appropriate error message and prompt the user again until the user makes a valid selection.

## Specifications

- Create an enumeration named Roshambo that stores three values: rock, paper, and scissors. This enumeration should include a toString method that can convert the selected value to a string.
- Create an abstract class named Player that stores a name and a Roshambo value. This class should include an abstract method named generateRoshambo that allows an inheriting class to generate and return a Roshambo value. It should also include get and set methods for the name and Roshambo value.
- Create classes named Bart and Lisa that inherit the Player class and implement the generateRoshambo method. The Bart class should always select rock. The Lisa class should randomly select rock, paper, or scissors (a 1 in 3 chance of each).
- Create a class named Player1 that inherits the Player class and implements the generateRoshambo method (even though it isn't necessary for this player). This method can return any value you choose.
- Create a class named RoshamboApp that allows the player to play Bart or Lisa as shown in the console output. Rock should beat scissors, paper should beat rock, and scissors should beat paper.
- Use the Validator class or a variation of it to validate the user's entries.

## Enhancement

- Keep track of wins and losses and display them at the end of each session.

# Chapter 10

## How to work with arrays

### Objectives

---

#### Applied

- Given a list of values or objects, write code that creates a one-dimensional array that stores those values or objects.
- Use for loops and enhanced for loops to work with the values or objects in an array.
- Use the methods of the `Arrays` class to fill an array, compare two arrays, sort an array, or search an array for a value.
- Implement the `Comparable` interface in any class you create.
- Create a reference to an array and copy elements from one array to another.
- Given a table of values or objects, write code that creates a two-dimensional array that stores those values or objects. The array can be either rectangular or jagged.
- Use for loops and enhanced for loops to work with the values or objects in a two-dimensional array.
- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

#### Knowledge

- In general, explain what an array is and how you work with it.
- Describe the operation of the enhanced for loop and explain why it's especially useful with arrays.
- Explain when you need to implement the `Comparable` interface in a class you create.
- Explain what happens when you assign a new array to an existing array variable.
- Describe the difference between a rectangular array and a jagged array, and explain the difference in how you create them.

#### Summary

---

- An *array* is a special type of object that can store more than one primitive data type or object. The *length* (or *size*) of an array is the number of *elements* that are stored in the array. The *index* is the number that is used to identify any element in the array.
- For loops are often used to process arrays. Java 5.0 also introduced a new type of for loop, called an *enhanced for loop* or a *foreach loop*, that lets you process each element of an array without using indexes.
- You can use the `Arrays` class to fill, compare, sort, and search arrays. You can use an assignment statement to create a second *reference* to the same array. And you can use the `arraycopy` method of the `System` class to make a copy of an array.

- To provide for sorting a user-defined class, that class must implement the Comparable interface.
- A *one-dimensional array* provides for a single list or column of elements so just one index value is required to identify each element. In contrast, a *two-dimensional array*, or an *array of arrays*, can be used to organize data in a table that has rows and columns. As a result, two index values are required to identify each element.
- A two-dimensional array can be *rectangular*, in which case each row has the same number of columns, or *jagged*, in which case each row has a different number of columns.

## Terms

---

array  
element  
length  
size  
index  
enhanced for loop  
foreach loop  
reference to an array  
one-dimensional array  
two-dimensional array  
array of arrays  
rectangular array  
jagged array

## Exercise 10-1 Use a one-dimensional array

---

In this exercise, you'll create an Array Test application so you can practice using one-dimensional arrays.

1. Open the ArrayTestApp class in the c:\java1.6\ch10 directory.
2. Create a one-dimensional array of 99 double values. Then, use a for loop to add a random number from 0 to 100 to each element in the array. For each value, use the random method of the Math class to get a double value between 0.0 and 1.0, and multiply it by 100.
3. Use an enhanced for loop to sum the values in the array. Then, calculate the average value and print that value on the console followed by a blank line. Compile and test this class.
4. Use the sort method of the Arrays class to sort the values in the array, and print the median value (the 50th value) on the console followed by a blank line. Then, test this enhancement.
5. Print the 9th value of the array on the console and every 9th value after that. Then, test this enhancement.

## Exercise 10-2 Use a rectangular array

---

This exercise will guide you through the process of adding a rectangular array to the Future Value application. This array will store the values for up to ten of the calculations that are performed, and print a summary of those calculations when the program ends that looks something like this:

Future Value Calculations			
Inv/Mo.	Rate	Years	Future Value
\$100.00	8.0%	10	\$18,416.57
\$125.00	8.0%	10	\$23,020.71
\$150.00	8.0%	10	\$27,624.85

Press any key to continue . . .

1. Open the FutureValueApp application stored in the c:\java1.6\ch10 directory.
2. Declare variables at the beginning of the main method for a row counter and a rectangular array of strings that provides for 10 rows and 4 columns.
3. After the code that calculates, formats, and displays the results for each calculation, add code that stores the formatted values as strings in the next row of the array (you need to use the toString method of the Integer class to store the years value).
4. Add code to display the elements in the array at the console when the user indicates that the program should end. The output should be formatted as shown above. Then, compile and test the program by making up to 10 future value calculations. When you've got this working right, close the program.

## **Exercise 10-3     Sort an array of user-defined objects**

---

In this exercise, you'll modify a `Customer` class so it implements the `Comparable` interface. Then, you'll sort an array of objects created from this class.

1. Open the `Customer` and `SortedCustomersApp` classes stored in the `c:\java1.6\ch10` directory.
2. Add code to the `Customer` class to implement the `Comparable` interface. The `compareTo` method you create should compare the email field of the current customer with the email field of another customer. To do that, you can't use the `>` and `<` operators because the email field is a string. Instead, you'll need to use the `compareToIgnoreCase` method of the `String` class. This method compares the string it's executed on with the string that's passed to it as an argument. If the first string is less than the second string, this method returns a negative integer. If the first string is greater than the second string, it returns a positive integer. And if the two strings are equal, it returns 0.
3. Add code to the `SortedCustomersApp` class that creates an array of `Customer` objects that can hold 3 elements, and create and assign `Customer` objects to those elements. Be sure that the email values you assign to the objects aren't in alphabetical order. Sort the array.
4. Code a `foreach` loop that prints the email, `firstName`, and `lastName` fields of each `Customer` object on a separate line.
5. Compile and test the program. When you're sure it works correctly, close the program.



## Exercise 10-4    Work with a deck of cards

---

In this exercise, you'll write an application that uses a variety of arrays and for loops to work with a deck of cards.

1. Open the CardDeckApp class in the `c:\java1.6\ch10` directory.
2. Create an array whose elements hold the first initial of the four different suits in a card deck. Declare an array that can hold a representation of the cards in a deck of cards without jokers.
3. Write a method to load the card array, one suit at a time. (Use the numbers 11, 12, and 13 to represent Jacks, Queens, and Kings respectively, and use the number 1 to represent Aces.) Write another method to print the cards in the array, separating each card by a space and printing each suit on a separate line. Call these two methods from the main method. Compile the application and test it to be sure the array is loaded properly.
4. Write a method that shuffles the deck of cards. To do that, this method should get a number between 1 and 51 by multiplying the result of the random function by 50, converting it to an integer, and adding 1. Then, it should switch each card in the deck with the card that is the given number of cards after it (if there is one). This should be repeated 100 times to shuffle the deck thoroughly. Call this method from the main method, followed by the method that prints the cards array. Test the application to be sure that the cards are shuffled.
5. Declare a rectangular array that represents four hands of cards with five cards each. Write a method that loads this array by dealing cards from the cards array. Be sure to deal one card at a time to each hand. Write a method that prints the hands, separating the cards in each hand by a space and printing each hand on a separate line. Test the application to be sure that the cards are dealt properly.

---

## Project 10-1: Calculate a player's batting statistics

---

### Console

```
Welcome to the Batting Average Calculator.

Enter number of times at bat: 5

0 = out, 1 = single, 2 = double, 3 = triple, 4 = home run
Result for at-bat 0: 0
Result for at-bat 1: 1
Result for at-bat 2: 0
Result for at-bat 3: 2
Result for at-bat 4: 3

Batting average: 0.600
Slugging percent: 1.200

Another batter? (y/n): y

Enter number of times at bat: 3

0 = out, 1 = single, 2 = double, 3 = triple, 4 = home run
Result for at-bat 0: 0
Result for at-bat 1: 4
Result for at-bat 2: 0

Batting average: 0.333
Slugging percent: 1.333

Another batter? (y/n): n
Press any key to continue . . .
```

### Operation

- This application calculates the batting average and slugging percentage for one or more baseball or softball players.
- For each player, the application first asks for the number of at bats. Then, for each at bat, the application asks for the result.
- To enter an at-bat result, the user enters the number of bases earned by the batter. If the batter was out, the user enters 0. Otherwise, the user enters 1 for a single, 2 for a double, 3 for a triple, or 4 for a home run.
- After all the at-bat results are entered, the application displays the batting average and slugging percent.

### Specifications

- The batting average is the total number of at bats for which the player earned at least one base divided by the number of at bats.
- The slugging percentage is the total number of bases earned divided by the number of at bats.
- Use an array to store the at-bat results for a player.

- Validate the input so the user can enter only positive integers. For the at-bat results, the user's entry must be 0, 1, 2, 3, or 4.
- Validate the user's response to the question "Another batter?" so the user can enter only Y, y, N, or n. If the user enters Y or y, calculate the statistics for another batter. Otherwise, end the program.
- Format the batting average and slugging percent to show three decimal digits.

## Enhancements

- At the start of the program, prompt the user for the number of batters to enter. Then, save the statistics for all of the batters in a two-dimensional array. The program won't have to ask the user whether to enter data for another batter, since it will know how many batters are to be entered. After all batters have been entered, print a one-line summary for each batter:

```
Batter 1 average: 0.357  slugging percent: 0.650  
Batter 2 average: 0.255  slugging percent: 0.550
```

- Instead of storing an array of integers, create a class named `AtBat` and store instances of this class in the array. This class should define an enumeration named `Result` with members `OUT`, `SINGLE`, `DOUBLE`, `TRIPLE`, and `HOMERUN`. The class should have a constructor that accepts a `Result` parameter and a method named `basesEarned` that returns an `int` representing the number of bases earned for the at bat.

---

## Project 10-2: Display a sorted list of student scores

---

### Console

```
Welcome to the Student Scores Application.

Enter number of students to enter: 4

Student 1 last name: Steelman
Student 1 first name: Andrea
Student 1 score: 95

Student 2 last name: Murach
Student 2 first name: Joel
Student 2 score: 92

Student 3 last name: Lowe
Student 3 first name: Doug
Student 3 score: 82

Student 4 last name: Murach
Student 4 first name: Mike
Student 4 score: 93

Lowe, Doug: 82
Murach, Joel: 92
Murach, Mike: 93
Steelman, Andrea: 95

Press any key to continue . . .
```

### Operation

- This application accepts the last name, first name, and score for one or more students and stores the results in an array. Then, it prints the students and their scores in alphabetical order by last name.

### Specifications

- The program should implement a class named `Student` that stores the last name, first name, and score for each student. This class should implement the `Comparable` interface so the students can be sorted by name. If two students have the same last name, the first name should be used to determine the final sort order.
- The program should use an array to store the `Student` objects. Then, it should sort the array prior to printing the student list.
- Validate the input so the user can enter only a positive integer for the number of students, the last or first name can't be an empty string, and the score is an integer from 0 to 100.

---

## Project 10-3: Display Quarterly Sales Report

---

### Console

```
Welcome to the Sales Report Application.

Region  Q1           Q2           Q3           Q4
1       $1,540.00     $2,010.00     $2,450.00     $1,845.00
2       $1,130.00     $1,168.00     $1,847.00     $1,491.00
3       $1,580.00     $2,305.00     $2,710.00     $1,284.00
4       $1,105.00     $4,102.00     $2,391.00     $1,576.00

Sales by region:
Region 1: $7,845.00
Region 2: $5,636.00
Region 3: $7,879.00
Region 4: $9,174.00

Sales by quarter:
Q1: $5,355.00
Q2: $9,585.00
Q3: $9,398.00
Q4: $6,196.00

Total annual sales, all regions: $30,534.00

Press any key to continue . . .
```

### Operation

- This application displays a four-section report of sales by quarter for a company with four sales regions (Region 1, Region 2, Region 3, and Region 4).
- The first section of the report lists the sales by quarter for each region.
- The second section summarizes the total annual sales by region.
- The third section summarizes the total annual sales by quarter for all regions.
- The fourth section prints the total annual sales for all sales regions.

### Specifications

- The quarterly sales for each region should be hard coded into the program using the numbers shown in the console output above. The sales numbers should be stored in a rectangular array.
- The first section of the report should use nested for loops to display the sales by quarter for each region. Use tabs to line up the columns for this section of the report.
- The second section of the report should use nested for loops to calculate the sales by region by adding up the quarterly sales for each region.
- The third section of the report should use nested for loops to calculate the sales by quarter by adding up the individual region sales for each quarter.
- The fourth section of the report should use nested for loops to calculate the total annual sales for the entire company.
- Use the `NumberFormat` class to format the sales numbers using the currency format.

## Enhancements

- Instead of hard-coding the data into the main program, have the program obtain the data from a static method of a separate class. For example, create a class called `SalesData` with a method called `getRegionSales`. This method would accept an `int` parameter representing the quarter, and return an array of doubles with quarterly sales for the specified region. The sales numbers could still be hard coded into this method.
- Alternatively, write the program so it prompts the user to enter the sales data. (The drawback to this approach is that testing the program will be tedious, as the student will have to retype all 16 sales numbers every time the program is run.)

## Chapter 11

# How to work with collections and generics

### Objectives

---

#### Applied

- Given a list of values or objects, write code that creates an array list or linked list to store the values or objects. Then, write code that uses the values or objects in the list.
- Given a list of key-value pairs, write code that creates a hash map or tree map to store the entries. Then, write code that uses the entries in the list.
- Given Java code that uses any of the language elements presented in this chapter, explain what each statement does.

#### Knowledge

- Describe the similarities and differences between arrays and collections.
- Name the two main types of collections defined by the collection framework and explain how they differ.
- Describe the generics feature and explain how you use it to create typed collections and classes.
- Explain what an array list is and, in general, how it works.
- Explain what autoboxing is.
- Explain what a linked list is and, in general, how it works
- Explain how you would decide whether to use an array list or a linked list for a given application.
- Explain what a queue is and describe the two basic operations that a queue provides.
- Describe the main difference between a hash map and a tree map.
- Explain what the legacy collections are.
- Explain what an untyped collection is and what you must do to work with one.
- Explain when you need to use a wrapper class with untyped collections.

## Summary

---

- A *collection* is an object that's designed to store other objects.
- The two most commonly used collection classes are `ArrayList` and `LinkedList`. An *array list* uses an array internally to store its data. A *linked list* uses a data structure with next and previous pointers.
- The *generics* feature, which became available with Java 5.0, lets you specify the type of elements a collection can store. This feature also lets you create *generic classes* that work with variable data types.
- A *map* is a collection that contains key-value pairs.
- The two most commonly used map classes are `HashMap` and `TreeMap`. The main difference between these two types of maps is that a *tree map* maintains its entries in key sequence and a *hash map* does not.
- Code that was written before Java 5.0 used *untyped collections*, which hold elements of type `Object`. To retrieve an element from an untyped collection, you typically have to use casting. To store primitive types in an untyped collection, you have to use *wrapper classes*.

## Terms

---

collection	array list
collection framework	autoboxing
set	linked list
list	queue
map	push operation
key-value pair	pull operation
generics	hash map
wrapper class	tree map
typed collection	legacy class
generic class	vector
type variable	untyped collection



## Exercise 11-1 Use an array list

---

This exercise will guide you through the process of adding a rectangular array to the Future Value application. This array will store the values for each calculation that is performed, and print a summary of those calculations when the program ends that looks something like this:

Future Value Calculations			
Inv/Mo.	Rate	Years	Future Value
\$100.00	8.0%	10	\$18,416.57
\$125.00	8.0%	10	\$23,020.71
\$150.00	8.0%	10	\$27,624.85

Press any key to continue . . .

1. Open the FutureValueApp class stored in the c:\java1.6\ch11\FutureValueArrayList directory.
2. Declare a variable at the beginning of the main method for an array list that stores strings.
3. After the code that calculates, formats, and displays the results for each calculation, add code that formats a string with the results of the calculation, then stores the string in the array list.
4. Add code to display the elements in the array list at the console when the user indicates that the program should end. Then, test the program by making at least 3 future value calculations.

## Exercise 11-2 Use a linked list

---

In this exercise, you'll modify the Future Value application you worked on in exercise 11-1 so it uses a linked list rather than an array list. In addition, you'll modify the code that displays the calculations so that the calculations are displayed in reverse order from the order in which they were entered.

1. Open the FutureValueApp class stored in the c:\java1.6\ch11\FutureValueArrayList directory and save it to the c:\java1.6\ch11\FutureValueLinkedList directory.
2. Change the variable declaration at the beginning of the main method from an array list to a linked list. Then, compile and test the application to see that it still works.
3. Modify the code that displays the calculations so it retrieves the elements of the linked list in reverse order. To do that, you'll need to use methods of the LinkedList class.
4. Compile and test the application again to be sure it works.

## Exercise 11-3 Create a stack

In this exercise, you'll create a class called `GenericStack` that uses a linked list to implement a stack, which is a collection that lets you access entries on a first-in, last-out basis. Then, you'll create another class that uses the `GenericStack` class. The `GenericStack` class should implement these methods:

Method	Description
<code>push(element)</code>	Adds an element to the top of the stack.
<code>pop()</code>	Returns and removes the element at the top of the stack.
<code>peek()</code>	Returns but does not remove the element at the top of the stack.
<code>size()</code>	Returns the number of entries in the stack.

### Create the `GenericStack` class

1. Start a new class named `GenericStack` that specifies a type variable that provides for generics. Then, save it in the `c:\java1.6\ch11\GenericStack` directory.
2. Declare a linked list that will hold the elements in the stack. Then, use the linked list to implement the methods shown above.
3. Compile the class.

### Create a class that uses the `GenericStack` class

4. Open the `GenericStackApp` class in the `c:\java1.6\ch11\GenericStack` directory.
5. Declare a generic stack at the beginning of the main method that will store `String` objects.
6. Add code to the main method that uses the `push` method to add at least three items to the stack. After each item is added, display its value at the console (you'll need to use a string literal to do this). Then, use the `peek` method to return the first item and display that item, and use the `size` method to return the number of items in the stack and display that value. Next, use the `pop` method to return each item, displaying it as it's returned. Finally, use the `size` method to return the number of items again and display that value.
7. Compile and run the class. If it works correctly, your output should look something like this:

```
Push: Apples
Push: Oranges
Push: Bananas
The stack contains 3 items

Peek: Bananas
The stack contains 3 items

Pop: Bananas
Pop: Oranges
Pop: Apples
The stack contains 0 items
```

---

## Project 11-1: List movies by category

---

### Console

```
Welcome to the Movie List Application.

There are 100 movies in the list.

What category are you interested in? scifi
Star Wars
2001: A Space Odyssey
E.T. The extra-terrestrial
A Clockwork Orange
Close Encounters Of The Third Kind

Continue? (y/n): y
What category are you interested in? comedy
Annie Hall
M*A*S*H
Tootsie
Duck Soup

Continue? (y/n): n
Press any key to continue . . .
```

### Operation

- This application stores a list of 100 movies and displays them by category.
- The user can enter any of the following categories to display the films in the list that match the category:
  - animated
  - drama
  - horror
  - scifi
- After each list is displayed, the user is asked whether to continue. If the user enters Y or y, the program asks for another category. Otherwise, the program ends.

### Specifications

- Each movie should be represented by an object of type `Movie`. The `Movie` class must provide two public fields: `title` and `category`. Both of these fields should be `Strings`. The class should also provide a constructor that accepts a `title` and `category` as parameters and uses the values passed to it to initialize its fields.
- You will be supplied with a class named `MovieIO` that has a method named `getMovie`. This method accepts an `int` argument that can be a number from 1 to 100. When called, it returns a unique `Movie` object for each value passed to it. You should use this method to fill the array list with 100 `Movie` objects.
- When the user enters a category, the program should read through all of the movies in the `ArrayList` and display a line for any movie whose category matches the category entered by the user.

## Enhancements

- Standardize the category codes by displaying a menu of category choices and asking the user to select the category by number rather than by entering the category code.
- Instead of an array list, use a tree map to store the movies. Then, display the movies for the selected category in alphabetical order.

---

## Project 11-2: Implement a stack calculator

---

### Console

```
Welcome to the Stack Calculator.

Commands: push n, add, sub, mult, div, clear, or quit.

? push 4
4.0

? push 3
3.0
4.0

? push 2
2.0
3.0
4.0

? mult
6.0
4.0

? add
10.0

? clear
empty

? quit

Thanks for using the Stack Calculator.

Press any key to continue . . .
```

### Operation

- This application implements a stack calculator that does arithmetic with doubles. It accepts commands in any of the following formats:

```
push double-value
```

```
add
```

```
sub
```

```
mult
```

```
div
```

```
clear
```

```
quit
```

## Specifications

- The calculator itself should be implemented as a separate class named `StackCalculator`. This class should have the following methods:

Method	Explanation
<code>void push(double x)</code>	Pushes <code>x</code> onto the stack.
<code>double x pop()</code>	Pops the top value from the stack.
<code>double add()</code>	Pops two values off the stack, adds them, and pushes the result back onto the stack.
<code>double subtract()</code>	Pops two values off the stack, subtracts them, and pushes the result back onto the stack.
<code>double multiply()</code>	Pops two values off the stack, multiplies them, and pushes the result back onto the stack.
<code>double divide()</code>	Pops two values off the stack, divides the first value into the second one, and pushes the result back onto the stack.
<code>void clear()</code>	Removes all entries from the stack.
<code>double[] getValues()</code>	Returns all of the values from the stack in array, without removing them from the stack.
<code>int size()</code>	Gets the number of values in the stack

- The `StackCalculator` class should use a linked list to maintain the stack data.
- The class that implements the user interface for the stack calculator should use a series of nested if statements in its main method to interpret the commands entered by the user.

## Hint

- You can use the `toArray` method of the `LinkedList` class to implement the `getValues` method. Although this method is shown in figure 11-9 in the book, that figure doesn't indicate that to use it with a typed collection, you must specify an array of the correct type as a parameter. See the online documentation for more information. (It's also possible to implement this method without using the `LinkedList`'s `toArray` method.)

## Enhancements

- Provide alternate forms for the commands that invoke operations. For example, allow `+` for add, `-` for sub, `*` for mult, and `/` for div.
- Add additional commands to the calculator. For example, `sqrt` could calculate the square root of the number on top of the stack, and `pow` could pop the top two values off the stack, raise the second value to the power indicated by the first value, and push the result back on the stack.
- Eliminate the need to use the word "push" to push data onto the stack. Instead, the calculator should treat any command that consists of just a number as a request to push the number onto the stack.

## Chapter 12

# How to work with dates and strings

### Objectives

---

#### Applied

- Use the `GregorianCalendar`, `Calendar`, `Date`, and `DateFormat` classes to get the current date, to set dates, to calculate elapsed days, and to format dates.
- Use the methods of the `String` class to manipulate and compare strings.
- Use the `StringBuilder` class to create a mutable string, and use the methods of a `StringBuilder` object to work with the string.
- Given Java code that uses any of the language elements presented in this chapter, explain what each statement does.

#### Knowledge

- Describe the difference between how dates are stored in `GregorianCalendar` and `Date` objects.
- Describe two situations where you would typically use `Date` objects rather than `GregorianCalendar` objects.
- Explain the difference between a mutable and an immutable string and why it's usually more efficient to use a mutable string.
- Explain how Java determines the initial capacity of a `StringBuilder` object and the new capacity of a `StringBuilder` object when its current capacity is exceeded.

### Summary

---

- You can use the `GregorianCalendar`, `Calendar`, `Date`, and `DateFormat` classes to create, manipulate, and format dates and times.
- You can use methods of the `String` class to locate a string within another string, return parts of a string, and compare all or part of a string. However, `String` objects are *immutable*, so you can't add, delete, or modify individual characters in a string.
- `StringBuilder` objects are *mutable*, so you can use the `StringBuilder` methods to add, delete, or modify characters in a `StringBuilder` object. Whenever necessary, Java automatically increases the capacity of a `StringBuilder` object.

### Terms

---

regular expression  
immutable string  
mutable string

## Exercise 12-1 Add a due date to the Invoice application

---

For this exercise, you'll modify the Invoice class that's shown in figure 12-6 so that it contains methods that return a due date, calculated as 30 days after the invoice date. Then, you'll modify the Invoice application that was shown in chapter 11 to display the invoice date and due date for a batch of invoices.

1. Open the Invoice and InvoiceApp classes in the c:\java 1.6\ch12\Invoice directory.
2. Add two methods named `getDueDate` and `getFormattedDueDate` to the Invoice class. The `getDueDate` method should calculate and return a Date object that's 30 days after the invoice date. The `getFormattedDueDate` method should return the due date in the short date format. Compile the class.
3. Modify the `displayInvoices` method in the InvoiceApp class so that the invoice display includes columns for the invoice date and the due date in addition to the invoice number and total. Then, compile this class and run it to make sure it works.

## Exercise 12-2 Calculate the user's age

---

In this exercise, you'll write a program that accepts a person's birth date from the console and displays the person's age in years. To make that easier to do, we'll give you a class that contains the code for accepting the birth date. The console output for the program should look something like this:

```
Welcome to the age calculator.  
Enter the month you were born (1 to 12): 5  
Enter the day of the month you were born: 16  
Enter the year you were born (four digits): 1959  
Your birth date is May 16, 1959  
Today's date is Sep 27, 2004  
Your age is: 45
```

1. Open the AgeCalculatorApp class in the c:\java 1.6\ch12\AgeCalculator directory.
2. Add code to this class that gets the current date and then uses the current year to validate the birth year the user enters. The user should not be allowed to enter a year after the current year or more than 110 years before the current year.
3. Add code to create, format, and print the user's birth date and to format and print the current date.
4. Add code to calculate and print the user's age.
5. Compile the class. Then, run it for a variety of dates to be sure it works.



### Exercise 12-3 Parse a name

---

In this exercise, you'll write an application that parses full names into first and last name or first, middle, and last name, depending on whether the user enters a string consisting of two or three words. The output for the program should look something like this:

```
Welcome to the name parser.  
  
Enter a name: Joel Murach  
  
First name:  Joel  
Last name:   Murach
```

1. Open the `NameParserApp` class in the `c:\java 1.6\ch12\NameParser` directory.
2. Add code to the main method that lets the user enter a full name as a string. Then, add code that separates the name into two or three strings depending on whether the user entered a name with two words or three. Finally, display each word of the name on a separate line. If the user enters fewer than two words or more than three words, display an error message. Also, make sure the application works even if the user enters one or more spaces before or after the name.
3. Compile the program and run it to make sure it works.

### Exercise 12-4 Validate a social security number

---

In this exercise, you'll add a method named `getSSN` to the `Validator` class that was presented in chapter 6. Then, you'll use this method in a program to validate a social security number entered by the user.

1. Open the `SSNValidatorApp` and `Validator` classes in the `c:\java1.6\ch12\SSNValidator` directory.
2. Add a method named `getSSN` to the `Validator` class that accepts and validates a social security number. This method should accept a `Scanner` object and a string that will be displayed to the user as a prompt. After it accepts the social security number, this method should validate the entry by checking that the number consists of three numeric digits, followed by a hyphen and two numeric digits, followed by a hyphen and four numeric digits. If the user's entry doesn't conform to this format, the method should display an error message and ask the user to enter the number again. Compile the class.
3. Modify the `SSNValidatorApp` class so that it uses the `getSSN` method. Then, compile and run this class to make sure the validation works correctly.

---

## Project 12-1: Calculate reservation totals

---

### Console

```
Welcome to the Reservation Calculator.

Enter the arrival month (1-12): 5
Enter the arrival day (1-31): 16
Enter the arrival year: 2005

Enter the departure month (1-12): 5
Enter the departure day (1-31): 18
Enter the departure year: 2005

Arrival Date: Monday, May 16, 2005
Departure Date: Wednesday, May 18, 2005
Price: $115.00 per night
Total price: $230.00 for 2 nights

Another reservation? (y/n): n
Press any key to continue . . .
```

### Operation

- This application calculates the charges for a stay at a hotel based on the arrival and departure dates.
- The application begins by prompting the user for the month, day, and year of the arrival and the departure.
- Next, the application displays the arrival date, the departure date, the room rate, the total price, and the number of nights.

### Specifications

- Create a class named `Reservation` that defines a reservation. This class should contain instance variables for the arrival date and departure date. It should also contain a constant initialized to the nightly rate of \$115.00.
- The `Reservation` class should contain a constructor that accepts the arrival and departure dates as parameters of type `Date`, as well as methods that return the number of nights for the stay (calculated by subtracting the arrival date from the departure date) and the total price (calculated by multiplying the number of nights for the stay by the nightly room rate). This class should also override the `toString` method to return a string like this:  

```
Arrival Date: Monday, May 16, 2005
Departure Date: Wednesday, May 18, 2005
Price: $115.00 per night
Total price: $230.00 for 2 nights
```
- The main method for the application class should contain a loop that asks the user for the arrival and departure date information, creates a `Reservation` object, and displays the string returned by the `toString` method.
- Assume valid data is entered.

## Enhancements

- Add validation so the user must enter values that will result in a correct date.
- Allow the user to enter the date in the form *mm/dd/yyyy*.
- Allow the user to enter the room rate or select the rate from one of several available rates.
- Use the `BigDecimal` class rather than the `double` type for the price calculation.

---

## Project 12-2: Translate English to Pig Latin

---

### Console

```
Welcome to the Pig Latin Translator.  
Enter a line to be translated to Pig Latin:  
this program translates from english to pig latin  
  
isthay ogrampray anslatestray omfray englishway otay igpay atinlay  
  
Translate another line? (y/n): n  
Press any key to continue . . .
```

### Operation

- The application prompts the user to enter a line of text.
- The application translates the text to Pig Latin and displays it on the console.
- The program asks the user if he or she wants to translate another line.

### Specifications

- Parse the string into separate words before translating. You can assume that the words will be separated by a single space and there won't be any punctuation.
- Convert each word to lowercase before translating.
- If the word starts with a vowel, just add *way* to the end of the word.
- If the word starts with a consonant, move all of the consonants that appear before the first vowel to the end of the word, then add *ay* to the end of the word.
- If a word starts with the letter *y*, the *y* should be treated as a consonant. If the *y* appears anywhere else in the word, it should be treated as a vowel.
- Check that the user has entered text before performing the translation.

### Enhancements

- Keep the case of the original word whether it's uppercase (TEST), title case (Test), or lowercase (test).
- Allow punctuation in the input string.
- Translate words with contractions. For example, *can't* should be *an'tcay*.
- Don't translate words that contain numbers or symbols. For example, 123 should be left as 123, and *bill@microsoft.com* should be left as *bill@microsoft.com*.

---

## Project 12-3: Convert numbers to words

---

### Console

```
Welcome to the Number to Word Converter.

Enter the number you want to convert to words: 3842
three thousand eight hundred forty two

Convert another number? (y/n): y

Enter the number you want to convert to words: 2001
two thousand one

Convert another number? (y/n): y

Enter the number you want to convert to words: 4815
four thousand eight hundred fifteen

Convert another number? (y/n): y

Enter the number you want to convert to words: 400
four hundred

Convert another number? (y/n): n

Press any key to continue . . .
```

### Operation

- The user enters a value from 0 to 9999, and the program converts it to an English representation of the value as shown in the console output.
- To program continues until the user responds to the “Convert another number” question with a value other than Y or y.

### Specifications

- You are free to use any technique you can devise to split the number entered by the user into its thousands, hundreds, tens, and ones digits and to create the string representation of the number.
- You can use arrays of String objects for units, teens, and tens words. For instance, the teens array may include {“ten”, “eleven”, “twelve”...and so on}. The tens array may include {“twenty”, “thirty”, “forty”...and so on}.
- If the tens digit is greater than 1, the word will use “twenty”, “thirty”, “forty”, and so on. Then, the last word be a units digit.
- If the number ends with 00, only the thousands and hundreds places are printed.
- If the number ends with 01 through 09, the last word will be a units digit.
- If the tens digit is 1, the last word will be “ten”, “eleven”, “twelve” and so on.

## Hints

- One way to split the user's input into separate digits is to treat the value as a string and use the substring method to extract the separate digits. If you use this technique, be sure to account for values with fewer than four digits.
- Another way to extract the separate digits is to treat the number as an integer and use a combination of integer division and modulo division. (Remember that integer division truncates the results.)
- Don't forget to provide for an entry of zero!

## Enhancements

- Validate the input to make sure it isn't negative or greater than 9,999.
- Allow negative numbers on input.
- Allow double input with up to two decimal digits, and format the string as it would be written on a check. (For example, "One hundred thirty two dollars and 38 cents.")

## Chapter 13

# How to handle exceptions

### Objectives

---

#### Applied

- Given a method that throws one or more exceptions, code a method that calls that method and catches the exceptions.
- Given a method that throws one or more exceptions, code a method that calls that method and throws the exceptions.
- Code a method that throws an exception.
- Use the methods of the `Throwable` class to get information about an exception.
- Code a class that defines a new exception, and then use that exception in an application.
- Use an `assert` statement in an application to make an assertion about a condition.
- Given Java code that uses any of the language elements presented in this chapter, explain what each statement does.

#### Knowledge

- Describe the `Throwable` hierarchy and the classes that are derived from it.
- Describe the difference between checked and unchecked exceptions and explain when you need to catch each.
- Explain how Java propagates exceptions and how it uses the stack trace to determine what exception handler to use when an exception occurs.
- Describe the order in which you code the catch clauses in a try statement.
- Explain what it means to swallow an exception.
- Explain when the code in the finally clause of a try statement is executed and how that compares to code that follows a try statement.
- Describe three situations where you might want to throw an exception from a method.
- Describe two situations where you might create a custom exception class.
- Explain what exception chaining is and when you might use it.
- Explain what assertions are and how you can use them in your applications.

## Summary

---

- In Java, an *exception* is an object that's created from a class that's derived from the Exception class or one of its subclasses. When an exception occurs, a well-coded program notifies its users of the exception and minimizes any disruptions or data loss that may result from the exception.
- Exceptions derived from the RuntimeException class and its subclasses are *unchecked exceptions* because they aren't checked by the compiler. All other exceptions are *checked exceptions*.
- Any method that calls a method that *throws* a checked exception must either throw the exception by coding a throws clause or *catch* it by coding *try/catch/finally blocks* as an *exception handler*.
- When coding your own methods, if you encounter a potential error that can't be handled within that method, you can code a *throw statement* that throws the exception to another method. If you can't find an appropriate exception class in the Java API, you can code your own exception class.
- You can create custom exception classes to represent exceptions your methods might throw. This is often useful to hide the details of how a method is implemented.
- When you use custom exceptions, you can use *exception chaining* to save information about the cause of an exception.
- An *assertion* lets you test that a condition is true at a specific point in an application.

## Terms

---

exception handling  
exception  
unchecked exception  
checked exception  
throw an exception  
catch an exception  
exception handler  
stack trace  
call stack  
swallowing an exception  
finally block  
exception chaining  
assertion



## Exercise 13-1 Research Java exceptions

---

Examine the Java API documentation and make a list of three unchecked exception classes and three checked exception classes that aren't listed in figure 13-1. For each exception, give a brief description of a programming situation in which you might want to catch the exception.

## Exercise 13-2 Throw and catch exceptions

---

In this exercise, you'll experiment with various types of exceptions and ways to handle them.

1. Open the `ExceptionTesterApp` class in the `c:\java1.6\ch13` directory. This class provides a framework you can use to experiment with exceptions thrown and caught at different levels of the call stack. It consists of a main method that calls a method named `Method1`, which in turn calls a method named `Method2`, which in turn calls a method named `Method3`. Each method displays a message before and after it calls the next method, and indentation is used to indicate the level in the call stack. Compile and run this class to get a feel for how it works.
2. Add code to `Method3` that throws an unchecked exception by attempting to divide an integer by zero. Compile and run the program and note where the exception is thrown.
3. Delete the code you just added to `Method3`. Then, add a statement to this method like the one in the first example in figure 13-5 that creates an object from the `RandomAccessFile` class. This class throws a checked exception named `FileNotFoundException`. Compile the class and note the error message that indicates that you haven't handled the exception.
4. Add the code necessary to handle the `FileNotFoundException` in `Method1`. To do that, you'll need to add `throws` clauses to the declarations of `Method2` and `Method3`. You'll also need to add a `try` statement to `Method1` that catches the exception. The catch block should display an error message. Run the program to make sure the exception handler works.
5. Remove the `try` statement from `Method1` and add a `throws` clause to the declarations for `Method1` and the main method. Then, run the program to see how a checked exception can propagate all the way out of a program.

### Exercise 13-3 Use the finally clause

---

In this exercise, you'll experiment with the finally clause to see how it works.

1. Open the `FinallyTesterApp` class in the `c:\java1.6\ch13` directory.
2. Modify the code in `Method3` so that it contains a try statement that includes code that throws an `IOException`, a catch clause that handles the exception, and a finally clause. Add a statement to each clause of the try statement that prints information to the console so you can trace the execution of the program. Run the program to make sure that the catch clause catches the exception and that the finally clause is executed as expected.
3. Add an if statement to the try block that throws a `NoSuchMethodException` if a condition is true and the `IOException` if the condition is false. Use any condition you want, but be sure that it evaluates to true so the `IOException` won't be thrown. Instead of adding a catch clause to catch the new exception, add throws clauses to the `Method1`, `Method2`, and `Method3` declarations so that the exception is thrown up to the main method.
4. Add a try statement to the main method, and call `Method1` from within the try clause. Then, add a catch clause that catches the `NoSuchMethodException` and displays a message indicating that the exception has been caught. Run the application to verify that the code in the finally clause in `Method3` is still executed, but the code that follows the try statement is not.

### Exercise 13-4 Create a custom class

---

In this exercise, you'll experiment with custom classes and chained exceptions.

1. Create a custom checked exception class named `TestException` that contains two constructors: one that accepts no parameters and one that accepts a `String` message.
2. Open the `CustomTesterApp` class in the `c:\java1.6\ch13` directory.
3. Add a statement to `Method3` that throws a `TestException` without a message. (You'll also need to delete or comment out the last statement in `Method3` or the compiler will flag it as unreachable.) Add the code necessary to catch this exception in `Method2`. The catch block should simply display a message at the console. Compile and run the program and observe its operation.
4. Modify your solution so that a custom message is passed to the `TestException` and is then displayed in the catch block. Compile and run the program to be sure that the custom message is displayed.
5. Add another constructor to the `TestException` class that accepts a `Throwable` object as a parameter.
6. Add a try statement to `Method3` of the `CustomTesterApp` class. The try clause should throw an `IOException`, and the catch clause should throw a `TestException`, passing the `IOException` to its constructor.
7. Modify the catch block in `Method2` that catches the `TestException` so that it displays the original cause of the exception. Compile and run the application to make sure it works.

## **Exercise 13-5    Use the assert statement**

---

In this exercise, you'll add an assert statement to the Invoice application of chapter 4 so you can see how it works.

1. Open the InvoiceApp class in the `c:\java1.6\ch13` directory. Notice that the statement that calculates the invoice total has been changed so that it adds the discount amount to the subtotal instead of subtracting it.
2. Add an assert statement that tests that the calculated invoice total is always less than or equal to the subtotal entered by the user. Include an appropriate message to be displayed if this assertion is false. Then, compile and run the application to see that this statement isn't executed by default.
3. Enable assertions, and then run the program again. This time, an assertion error should occur and the message you specified should be displayed.

---

## Project 13-1: Check if a file exists

---

### Console

```
Welcome to the File Checker Application.

Enter a file path and name: c:\autoexec.bat
That file exists.

Check another file? (y/n): y

Enter a file path and name: c:\Murach\ASP.NET\Student workbook.pdf
That file exists.

Check another file? (y/n): y

Enter a file path and name: c:\badfile.dat
That file does not exist.

Check another file? (y/n): n
Press any key to continue . . .
```

### Operation

- The user enters a file path and name, and the program checks to see if the file exists. If it does, it displays the message “That file exists.” If it doesn’t, it displays the message “That file does not exist.” The program then asks if the user wants to check another file.

### Specifications

- Create a method named `doesFileExist` that accepts a `String` argument for a file path and name and returns a boolean value to indicate whether or not the file exists.
- Use the `FileInputStream` class to determine if the file exists. The constructor for this class accepts a `String` that contains a file path as a parameter and throws a `FileNotFoundException` if the file doesn’t exist. The `doesFileExist` method should catch this exception to decide whether it should return `true` or `false`.
- The `FileInputStream` and `FileNotFoundException` classes are both in the `java.io` package.
- The main method should prompt the user for a file path and name, call the `doesFileExist` method, and display one of the two messages depending on whether `doesFileExist` returns `true` or `false`.

---

## Project 13-2: Display customer information

---

### Console

```
Welcome to the Customer application

Enter a customer number: 1003

Ronda Chavan
518 Commanche Dr.
Greensboro, NC 27410

Display another customer? (y/n): y

Enter a customer number: 2439

The customer 2439 does not exist.

Display another customer? (y/n): n

Press any key to continue . . .
```

### Operation

- This application displays customer information for customers selected by the user. The application prompts the user to enter a customer number. If a customer with that number exists, the application displays the customer's name and address. If no customer with that number exists, the application displays the message "The customer *number* does not exist." Either way, the application then asks if the user wants to display another customer.

### Specifications

- Create a class named `Customer` that stores name, address, city, state, and zipCode as public fields. The class should have a method named `getNameAndAddress` that returns the name and address information formatted as shown in the console output above.
- To get the information for a customer, use the `CustomerIO` class that's provided. This class contains a method named `getCustomer` that accepts a customer number (an int value) and returns a `Customer` object.
- Modify the `getCustomer` method so that if it's called with an invalid customer number, it throws an exception of type `NoSuchCustomerException`.
- Create a `NoSuchCustomerException` class. This class should have a constructor that accepts an int parameter that provides the customer number that doesn't exist. This constructor should pass the message "The customer *number* does not exist." To the constructor of the Exception class.

- The `NoSuchCustomerException` should store the customer number as a private instance variable and make it available through a method named `getCustomerNumber`. If the user enters an invalid customer number, the main application class should use the `getCustomerNumber` method to retrieve the customer number from the exception object when it displays the error message to the user.

## Enhancements

- Provide a constructor for the `Customer` class that accepts a customer number as a parameter and then attempts to create a `Customer` object using the data for that customer. Then, handle `NoSuchCustomerException` in the constructor of the `Customer` class. The constructor should throw the `NoSuchCustomerException` if an invalid customer number is passed to it.
- Same as the first enhancement, but have the constructor throw a different exception named `CouldNotCreateCustomerException` if it can't create the customer. Then, it should chain the original `NoSuchCustomerException` in the `CouldNotCreateCustomerException`.
- Enhance the `getCustomer` method so that it also throws `IOException`. To simulate the random nature of `IOExceptions`, have your students code the class so that `IOException` is thrown randomly, with a 10% chance of the exception being thrown.

## Chapter 14

# How to work with threads

### Objectives

---

#### Applied

- Use the Thread class or the Runnable interface to create a thread.
- Use the methods of the Thread class to control when the processor executes a thread.
- Use the interrupt method and the InterruptedException class to create a thread that can be interrupted.
- Use the synchronized keyword to create synchronous threads.
- Use the wait and notifyAll methods of the Object class to coordinate the execution of two interdependent threads.

#### Knowledge

- Explain the basic difference between a program that runs in a single thread and a program that runs under multiple threads.
- Name three common reasons for using threads in a Java application.
- List the three Java API classes or interfaces that have methods related to threading.
- Explain the advantage of creating a thread by extending the Runnable interface rather than by inheriting the Thread class.
- List the five states of a thread, and describe the status of a thread in each state.
- Explain the difference between the sleep and yield methods.
- Explain why methods that can be executed concurrently by multiple threads need to be synchronized.
- Describe the producer/consumer pattern used for concurrency control.

#### Summary

---

- A *thread* is a single sequential flow of control within a program that often completes a specific task.
- A *multithreaded application* consists of two or more threads whose execution can overlap.
- Since a processor can only execute one thread at a time, the *thread scheduler* determines which thread to execute.
- *Multithreading* is typically used to improve the performance of applications with I/O operations, to improve the responsiveness of GUI operations, and to allow two or more users to run server-based applications simultaneously.

- You can create a thread by extending the Thread class and then instantiating the new class. Or, you can implement the Runnable interface and then pass a reference to the Runnable object to the constructor of the Thread class.
- You can use the methods of the Thread class to start a thread, to control when a thread runs, and to control when other threads are allowed to run.
- *Synchronized methods* can be used to ensure that two threads don't run the same method of an object simultaneously. When a thread calls a synchronized method, the object that contains that method is *locked* so that other threads can't access it.

## Terms

---

thread  
main thread  
multithreading  
central processing unit (CPU)  
servlet  
thread scheduler  
daemon thread  
user thread  
asynchronous threads  
synchronous threads  
concurrency  
locking  
synchronized method  
producer/consumer pattern  
producer  
consumer



## Exercise 14-1 Create a Number Finder application

---

In this exercise, you'll create an application that generates a random number between 0 and 999, and then uses four threads to search for the number. When one of the threads finds the number, it should print a message on the console. The output from this application should look like this:

```
The number is 784
Target number 784 found by Thread-3
```

1. Open the `NumberFinderThreadApp` class in the `c:\java1.6\ch14` directory. Review its code to see that it generates a random number between 0 and 999 and then displays it at the console.
2. Add a class named `Finder` that extends the `Thread` class to the `NumberFinderThreadApp` file. Then, add a constructor to this class that accepts three parameters: the number to search for, the number where the search should begin, and the number where the search should end.
3. Add a `run` method to the `Finder` class that searches for the number. This method should use a `for` loop to check each value in the specified range to determine if it matches the target value. If a match is made, the thread should display a message like the one shown above and terminate. Every ten times through the loop, the thread should yield to other threads.
4. Add code to the main method to create and start the four threads. The threads should check the following ranges: `thread0`, 0-249; `thread1`, 250-499; `thread2`, 500-749; and `thread3`, 750-999.
5. Compile the program. Then, run it two or more times to be sure it works correctly.

## Exercise 14-2 Use the Runnable interface and the sleep method

---

In this exercise, you'll modify your solution to exercise 14-1 so that the `Finder` class implements the `Runnable` interface instead of extending the `Thread` class and so that it calls `sleep` rather than `yield` every ten times through the loop.

1. Open the `NumberFinderThreadApp` class you worked on in exercise 14-1. Then, change the class name to `NumberFinderRunnableApp` and save the class with this file name.
2. Modify the `Finder` class so it uses the `Runnable` interface. Then, compile the program and run it to make sure it works correctly.
3. Modify the `Finder` class so it uses the `sleep` method to cause the thread to sleep for 1 millisecond every ten times through the loop. Then, compile and test the program again.

## Exercise 14-3 Add a Monitor thread to the Number Finder application

---

Because this exercise requires the use of a collection, you need to read chapters 10 and 11 before you do it. In this exercise, you'll modify your solution to exercise 14-2 so that the thread that finds the number notifies a Monitor thread, which then interrupts all of the Finder threads. When a Finder thread is interrupted, it should display a line indicating that it has been interrupted and then end. The resulting output should look like this:

```
The number is 20
Target number 20 found by Thread-1
Thread-2 interrupted
Thread-3 interrupted
Thread-4 interrupted
```

1. Open the `NumberFinderRunnableApp` class you worked on in exercise 14-2. Then, change the class name to `NumberFinderMonitorApp` and save the class with that file name.
2. Add a class named `Monitor` to the `NumberFinderRunnableApp` file. This class should define a thread by extending the `Thread` class. This class should include a method named `addThread` that adds a thread to a private collection of `Thread` objects. (You choose the type of collection.) It should also include a synchronized method named `foundNumber` that interrupts each thread in the threads collection. This method should also set a boolean instance variable to true to indicate that the number has been found. Then, the run method of this class can simply test this variable within a never-ending loop.
3. Modify the application's main method so that it creates and starts the `Monitor` thread, passes a reference to the `Monitor` thread to the Finder threads, and adds the four Finder threads to the `Monitor` thread.
4. Modify the `Finder` class so that its constructor accepts a reference to the `Monitor` thread. Then, modify the run method so that it calls the `Monitor` thread's `foundNumber` method if it finds the target number. Also modify this method so that a message is displayed when the thread is interrupted. Keep in mind that the thread may still be searching for the number when it's interrupted or it may have finished its search.
5. Compile the application and run it to be sure that it works correctly.

---

## Project 14-1: Tortoise and the hare race

---

### Console

```
Get set...Go!
Tortoise : 10
Tortoise : 20
Tortoise : 30
Tortoise : 40
Hare : 100
Tortoise : 50
Tortoise : 60
Tortoise : 70
Tortoise : 80
Hare : 200
Tortoise : 90
Tortoise : 100
.
.   (output lines omitted)
.
Hare : 500
Tortoise : 900
Tortoise : 910
Tortoise : 920
Tortoise : 930
Tortoise : 940
Tortoise : 950
Tortoise : 960
Tortoise : 970
Tortoise : 980
Tortoise : 990
Tortoise : 1000
Tortoise: I finished!

The race is over! The Tortoise is the winner.

Hare: You beat me fair and square.

Press any key to continue . . .
```

### Operation

- This application simulates a race between two or more runners. The runners differ in their speed and how often they need to rest. One of the runners, named “Tortoise,” is slow but never rests. The other runner, named “Hare,” is ten times as fast but rests 90% of the time.
- There is a random element to the runners’ performance, so the outcome of the race is different each time the application is run.
- The race is run over a course of 1000 meters. Each time one of the runners moves, the application displays the runner’s new position on the course. The first runner to reach 1000 wins the race.
- When one of the runners finishes the race, the application declares that runner to be the winner and the other runner concedes.

## Specifications

- Each runner should be implemented as a separate thread using a class named `ThreadRunner`. The `ThreadRunner` class should include a constructor that accepts three parameters: a string representing the name of the runner, an int value from 1 to 100 indicating the likelihood that on any given move the runner will rest instead of run, and an int value that indicates the runner's speed—that is, how many meters the runner travels in each move.
- The `run` method of the `ThreadRunner` class should consist of a loop that repeats until the runner has reached 1000 meters. Each time through the loop, the thread should decide whether it should run or rest based on a random number and the percentage passed to the constructor. If this random number indicates that the runner should run, the class should add the speed value passed to the constructor. The `run` method should sleep for 100 milliseconds on each repetition of the loop.
- If the `run` method is interrupted, it should display a message that concedes the race and quits.
- The main method of the application's main class should create two runner threads and start them. One of the threads should be named "Tortoise." It runs only 10 meters each move, but plods along without ever resting. The other thread should be named "Hare." It should run 100 meters each move, but should rest 90% of the time.
- This class should also include a method named `finished` that one of the threads can call when it finishes the race. That method should declare the thread that calls it to be the winner and should interrupt the other thread so it can concede the race.
- The `finished` method should provide for the possibility that the two threads will finish the race at almost the same time. If that happens, it should ensure that only one of the threads is declared the winner. (There are no ties!)

## Hints

- To determine whether a thread should run or rest, calculate a random number between 1 and 100. Then, have the thread rest if the number is less than or equal to the percentage of time that the thread rests. Otherwise, the thread should run.
- The `finished` method in the main application class will need to know which thread called it.

## Enhancements

- Modify the main application class so that it runs the race 100 times and reports how many times each runner wins. (To make the application run faster, you may want to reduce the sleep time in the runner threads.)
- Modify the application so it can support up to 9 runners.
- Add an additional random element to the runner's performance. For example, have a "clumsiness percentage" that indicates how often the runner trips and hurts himself. When the runner trips, he sprains his or her ankle and can run only at half speed for the next five moves.
- Add the ability for runners to interfere with each other. For example, have an "orneriness percentage" that indicates how likely the runner is to trip another runner who is passing him. This will require additional communication among the threads.



- It's difficult to tell while the application is running, but the streamers aren't displayed on the same line of console output. Instead, the console lines alternate between the first streamer and the second streamer.
- The effect of this animation is difficult to describe. To see it in action, you can download the ThreadProject.jar file for this application by opening your web browser, entering [www.murach.com/downloads/jav5/ThreadProject.jar](http://www.murach.com/downloads/jav5/ThreadProject.jar), and clicking the Save button in the dialog box that's displayed. Then, you can run the application by opening a command window, using the cd command to change to the directory where you saved the jar file, and typing the following command:

```
java -jar ThreadProject.jar
```

To stop the application, press Ctrl+C.

### Specifications

- Each streamer should be drawn by a separate thread. The threads should be created from a class named ThreadAnimator that accepts two parameters: an int that represents the length of time the thread should sleep after drawing each line, and a boolean that indicates whether the streamer should start at the left edge of the console (true) or the right edge (false).
- The threads should run indefinitely. The only way to stop the program is to close the console window.

### Hints

- To prevent the output from the threads from being mixed up, the code that generates the output will need to be placed in a synchronized method.
- You can vary the sleep time to speed up or slow down the animation. If the animation isn't working right, slowing it down may help you find out why.

### Enhancements

- Provide a way to terminate the thread by entering a command such as "stop" at the console.
- Modify the streamers so that they bounce off each other when they meet at the middle of the console.
- Add additional streamers. You'll have to modify the code to allow a streamer to start in the middle of the console rather than at one of the edges.

# Chapter 15

## How to get started with Swing

### Objectives

---

#### Applied

- Use the `JFrame` class to create and display an empty frame with a specified size and title, either at a specific position or centered on the screen.
- Set the default close behavior of a frame.
- Use the `JPanel` class to create a panel that contains labels, text fields, and buttons and add the panel to a frame.
- Use the `ActionListener` interface to provide basic event handling for `JButton` components.
- Use the `FlowLayout` and `BorderLayout` managers to control the layout of components on a panel.
- Given the requirements for a program that uses the Swing features presented in this chapter, develop a program that satisfies the requirements.

#### Knowledge

- Explain the difference between a frame and a panel.
- Describe the characteristics of the following types of controls: labels, text fields, and buttons.
- Describe the difference between AWT and Swing.
- List the classes that are in the inheritance hierarchy for all Swing component classes.
- Explain why it is necessary to set the default close behavior for a frame.
- List the four panes that are present in every frame and identify the pane that contains components such as text fields, labels, and buttons.
- Describe how button click events are handled in a Swing application.
- Explain the role of layout managers in Swing development, and describe the difference between the `FlowLayout` and `BorderLayout` managers.

#### Summary

---

- You can use *Swing* components to create graphical user interfaces that are platform independent and more bug-free than GUIs developed with the older GUI technology known as the *Abstract Window Toolkit (AWT)*.
- All Swing classes inherit the `Component` and `Container` classes, are stored in the `javax.swing` package, and begin with the letter J. Since all Swing components inherit the `Component` class, you can call any methods of the `Component` class from any Swing component.

- You can use Swing components to create a *frame* that contains a title bar and a border. Then, you can add *panels*, *labels*, *text fields*, and *buttons* to the *content pane* of that frame.
- You can use the Toolkit class to get the height and width of a user's screen in *pixels*. Then, you can use this information to center the frames you create on the screen.
- When coding a graphical user interface, you write code that handles *events* that are initiated by the user. To do that, you must write code that defines a *listener* that listens for each event and responds when an event occurs.
- You can use *layout managers*, such as the *Flow layout manager* and the *Border layout manager*, to control how components are displayed within a frame or panel. When using these layout managers, it's common to nest one panel within another panel.

## Terms

---

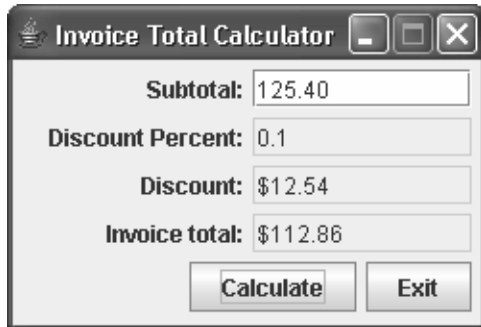
console application	Abstract Window Toolkit (AWT)
graphical user interface (GUI)	heavyweight component
Swing	lightweight component
Swing classes	container
Swing set	pixel
frame	thread
title bar	toolkit
panel	pane
label	content pane
text field	event listener
button	action event
component	layout manager
focus	Flow layout manager
Metal look and feel	Border layout manager



## Exercise 15-1 Create a Swing version of the Invoice application

---

In this exercise, you'll create a Swing version of the Invoice application that was presented in figure 2-18 of chapter 2. The application should have a user interface that looks something like this:



1. Open the InvoiceApp.java file in the c:\java1.6\ch15 directory. This file contains a public InvoiceApp class with an empty main method.
2. Add a class that defines the frame shown above. This frame should be centered on the screen, it should not be resizable, and closing it should end the application. This frame should be displayed when the application starts.
3. Add a class that defines a panel with the controls shown above, using a Flow layout manager to align the controls. Implement the ActionListener interface for this class so it will respond to the user selecting the Exit or Calculate button. If the user selects the Exit button, the application should end. If the user selects the Calculate button, the application should calculate and display the discount percent, discount amount, and invoice total. (The discount percent should be 20% if the subtotal is greater than or equal to \$200, 10% if the subtotal is less than \$200 but greater than or equal to \$100, and 0% if the subtotal is less than \$100.) Create an instance of this class and add it to the frame from the frame class.
4. Add the import statements needed by this application. Then, compile the application and run it to be sure it works correctly.

## **Exercise 15-2    Enhance the Invoice application**

---

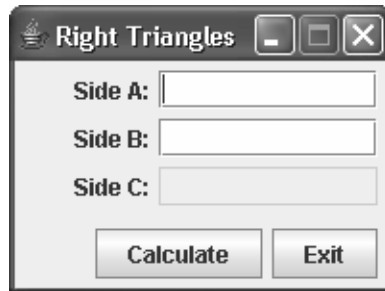
In this exercise, you'll enhance the Invoice application that you created for exercise 15-1.

1. Open the InvoiceApp.java file in the c:\java1.6\ch15 directory.
2. Modify the Invoice application so it uses three panels. The main panel should use Border layout and should serve only as a container for the other two panels. The labels and text fields should be added to a second panel that uses Flow layout, and the buttons should be added to a third panel that uses Flow layout. Add the second panel to the center region of the main panel and add the third panel to the south region of the main panel.
3. Modify the actionPerformed method so that if the value entered by the user can't be converted to a valid number, the application clears all the text fields and doesn't perform any calculations. To do that, you need to use a try/catch statement within this method.
4. Add a third button labeled Clear to the left of the other buttons. If the user selects this button, the application should clear the contents of all four text fields.
5. Compile the application and test it to be sure these changes work.

---

## Project 15-1: Calculate the length of a right triangle's hypotenuse

---

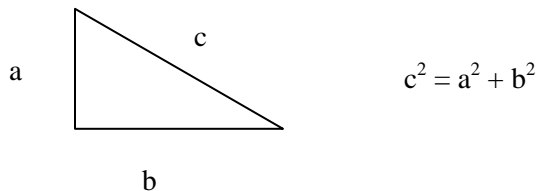


### Operation

- This application lets the user enter the lengths of the two shortest sides of a right triangle. When the user clicks Calculate, the program calculates and displays the length of the third side.

### Specifications

- Use the Pythagorean Theorem to calculate the length of the third side. The Pythagorean Theorem states that the square of the hypotenuse of a right-triangle is equal to the sum of the squares of the opposite sides:



- Don't worry about validating the user's input. If the user enters non-numeric data or fails to enter data, allow the program to fail.

### Hint

- If you have trouble getting the labels and text fields to line up properly, try adjusting the frame size. When you use the Flow layout manager, the width of the frame affects how components you add to the frame are lined up.

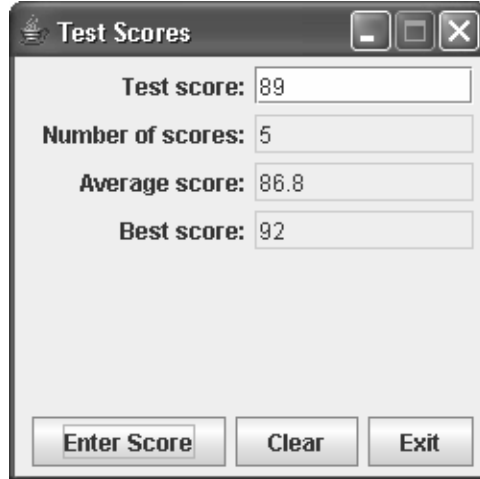
### Enhancements

- Add data validation by catching the exception that's thrown when the user enters invalid data.
- Research the Graphics class or the Java2D API and add a drawing that represents the triangle specified by the user. You may want to scale the triangle's dimensions to fit within a pre-defined area of the frame.

---

## Project 15-2: Accumulate test scores

---



### Operation

- The user enters test scores one at a time and then clicks the Enter Score button.
- For each entered score, the application adds one to the number of scores, calculates the average score, and determines what the best score is so far. Then, it displays the number of scores, average score, and best score in the three disabled text fields.
- The user can click the Clear button to reset everything to zero.
- When the user closes the frame or clicks the Close button, the application exits.

### Specifications

- The average score is the sum of all scores divided by the number of scores.
- Assume valid data is entered.

### Hint

- If you have trouble getting the labels and text fields to line up properly, try adjusting the frame size. When you use the Flow layout manager, the width of the frame affects how components you add to the frame are lined up.

### Enhancement

- Add data validation by catching the exception that's thrown when the user enters invalid data.

## Chapter 16

# How to work with controls and layout managers

### Objectives

---

#### Applied

- Create a panel that includes a text area with a scroll bar.
- Create a panel that includes check boxes and radio buttons, and write an event listener that responds to click events for the controls.
- Create a border that includes both a line style such as etched or beveled and a title. Then, apply this border to a group of check boxes, radio buttons, or other controls.
- Create a panel that includes a combo box or list populated with data from an array or collection, and write an event listener that responds when the user selects an item.
- Create an event handler that can process all of the items selected in a list that allows multiple selections.
- Create a panel that includes a list whose contents can be changed as the program executes.
- Given a desired layout for a panel, draw a grid that represents the layout. Then, use the Grid Bag layout manager to build the panel with the correct layout.
- Given the requirements for a program that uses text areas, scroll panes, check boxes, radio buttons, borders, combo boxes, and lists, write the code to implement the application.

#### Knowledge

- List two Swing components that are designed to enhance the appearance or operation of other controls.
- Describe a situation in which you would use a text area rather than a text field.
- List two ways a program can determine whether a user has selected a check box or radio button.
- Explain the difference between `ActionEvent` and `ItemEvent` for a combo box.
- Explain the difference between a combo box and a list.
- List six layout managers commonly used to build Swing applications, and describe the approach each takes to arranging controls in a panel.

## Summary

---

- You can create a *text area* that can store one or more lines of text, and you can use many of the same techniques to work with text fields and text areas.
- You can create two or more *radio buttons* that you can add to a *button group*. Then, the user can select one of the buttons in the group. You can also create a *check box* that lets a user check or uncheck the box.
- A *combo box* lets a user select an item from a drop-down list of items, and a *list* lets a user select one or more items from a list of items.
- You can add a component like a text area or list to a *scroll pane*, and you can add a *border* to any component.
- The *Grid Bag layout manager* is the most sophisticated and flexible layout manager. When you use the Grid Bag layout manager, you use the fields of the `GridBagConstraints` class to position components in a grid.

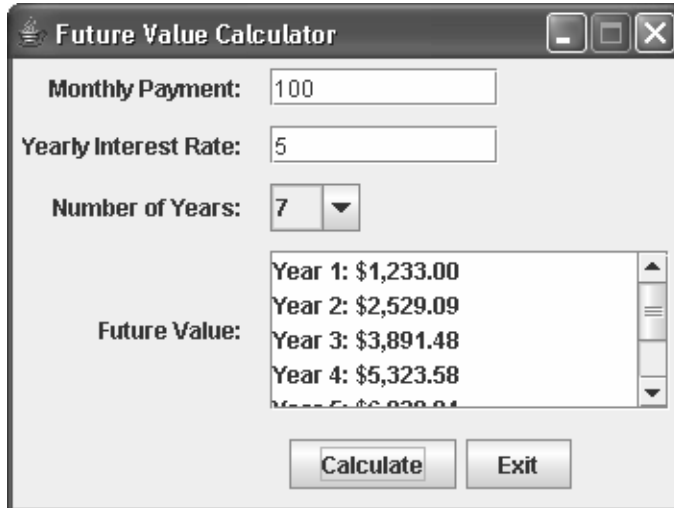
## Terms

---

text area	item event
scroll pane	action event
check box	list
radio button	selection mode
button group	multiple interval selection
border	list model
combo box	Grid Bag layout manager

## Exercise 16-1    **Modify the Future Value application**

For this exercise, you'll modify the Future Value application presented in chapter 15 so that it uses a combo box, a list, and the Grid Bag layout manager. When you're done, the user interface should look something like this:



1. Open the FutureValueApp class in the c:\java1.6\ch16\FutureValue directory.
2. Modify the panel so that it uses the Grid Bag layout manager rather than the Flow layout manager. If you want to, you can use the `getConstraints` method that's shown in part 3 of figure 16-15 to help set the grid bag constraints.
3. Replace the Number of Years text field with a combo box that contains the values 1 through 20.
4. Replace the Future Value text field with a list that displays five rows and uses a vertical scroll bar.
5. Modify the action event listener for the Calculate button so that instead of calculating a single future value, it calculates the future value for each year up to the year selected via the combo box and adds a string showing the calculation for each year to the list.
6. Compile the program, then test it to be sure it works correctly.

## Exercise 16-2 Create a Pizza Calculator application

For this exercise, you'll develop an application that calculates the price of a pizza based on its size and toppings. The user interface for this application should look something like this:



1. Decide what layout manager or combination of layout managers you want to use to implement the user interface, and then sketch the user interface and its rows and columns.
2. Open the `PizzaOrderApp.java` file in the `c:\java1.6\ch16\PizzaOrder` directory. This file contains a public `PizzaOrderApp` class with an empty main method.
3. Add the code necessary to implement this application. When the user selects a size and toppings for the pizza and clicks the `Calculate` button, the application should calculate the price of the pizza and display that price in the text field. To calculate the price of the pizza, add the price of the selected toppings to the base price of the pizza:

Item	Price
Small pizza	\$6.99
Medium pizza	\$8.99
Large pizza	\$10.99
Sausage	\$1.49
Pepperoni	\$1.49
Salami	\$1.49
Olives	\$0.99
Mushrooms	\$0.99
Anchovies	\$0.99

4. Compile the program and test it to be sure it works correctly.



## Project 16-1: Enter a team lineup

### Console output

```
Rage 12U (home team)
H. Perkins, Center Field
C. Cousins, Pitcher
J. Johnson, Catcher
B. Lowe, First Base
A. Licouris, Left Field
N. Shrey, Short Stop
S. Palmore, Third Base
C. Giess, Second Base
A. Nieto, Right Field
```

### Operation

- This application lets the user enter the batting lineup for a baseball or softball team.
- The user can enter a team name and use the radio buttons to select whether the team is the home team or the visiting team.
- For each player, the user enters the player's name in the text field and selects the player's position from the combo box.
- When the user clicks OK, the application displays the team roster on the console as shown in the console output.

## Specifications

- The team positions in the combo boxes should offer the following choices:
- Choose a selection
- Pitcher
- Catcher
- First base
- Second base
- Third base
- Short stop
- Left field
- Center field
- Right field
- Don't worry about validating the user's input.
- Use the Grid Bag layout to control the positioning of the labels, text fields, combo boxes, and buttons.

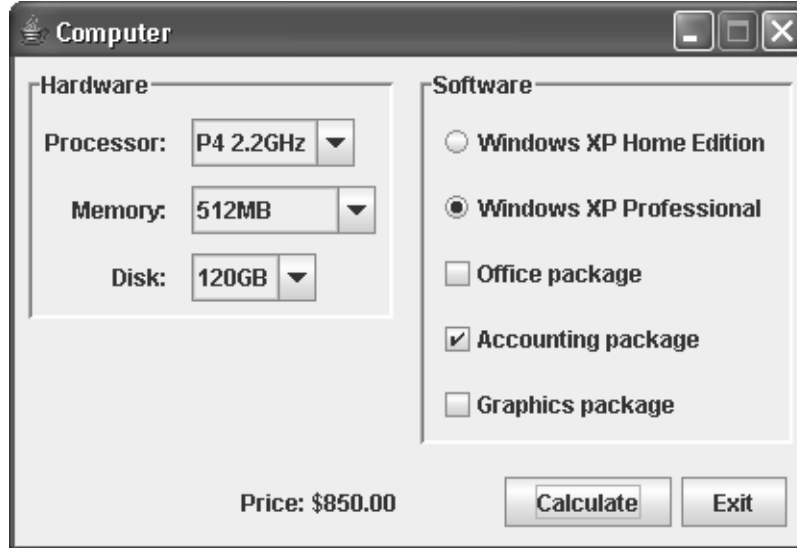
## Enhancements

- Add data validation by requiring that all text fields be entered.
- Validate the combo box selections so that the user can't select the same position for two or more players. If the user attempts to set a combo box to a selection that's already been assigned to a player, the combo box should immediately revert to "Choose a position."
- Replace the text field for each player with a combo box that lists all of the players on the team. Fill these combo boxes with a team roster consisting of at least 9 names, and do not allow the user to place the same player in more than one spot in the lineup. You can either initialize the roster with names coded as constants in the program, or you can read the names from a text file if you've already taught file handling. (You could also supply the students with a class that has a method that returns an array list with the names.)

---

**Project 16-2: Configure a computer purchase**

---

**Operation**

- The user configures the components of a computer system by selecting items from the combo boxes, radio buttons, and check boxes.
- When the user clicks the Calculate button, the application calculates the price of the system by adding the cost of each selected component to a base price.
- When the user closes the frame or presses the Exit button, the application exits.

**Specifications**

- You are free to use whatever layouts and panels you need to duplicate the layout shown in the figure. You can also create any classes you think might be helpful to solve this problem.
- The base price of the computer is \$500.
- The user can select one of three processors:

P4 2.2GHz	included in base price
P4 2.4GHz	add \$50.00
P4 2.6GHz	add \$150
- The user can select one of four memory configurations:

256MB	included in base price
512MB	add \$50.00
1GB	add \$100.00
2GB	add \$150.00

- The user can select one of three disk configurations:

80GB	included in base price
120GB	add \$50.00
170GB	add \$150.00
- The user can select one of two operating systems:

Windows XP Home Edition	included in base price
Windows XP Professional	add \$100.00
- The user can select any or all of the following three software packages:

Office package	add \$400.00
Accounting package	add \$200.00
Graphics package	add \$600.00

### Hints

- Consider using a separate class to represent each of the configuration options.
- Don't forget to calculate the initial price (\$500) when the program starts.

### Enhancements

- Add a feature that indicates how changing each selection will affect the price. For example, if Windows XP Professional is currently selected, change the text for the Windows XP Home Edition label to "Windows XP Home Edition (subtract \$100)." However, if the home edition is currently selected, the check box for the professional edition should read Windows XP Professional (add \$100)."
- Print a summary of the configuration on the console when the user clicks the Calculate button.
- If you've already covered file handling, retrieve the configuration options from a file rather than having them hard coded in the program.

## Chapter 17

# How to handle events and validate data

### Objectives

---

#### Applied

- Given a class that extends `JPanel` and includes one or more components that generate events, modify the class so that it implements the appropriate listener interface to respond to those events.
- Given a class that extends `JPanel` and includes one or more components that generate events, create a separate class to handle those events, and modify the panel class so that it adds the event class as a listener for each event source.
- Given a class that extends `JPanel` and includes one or more components that generate events, add an inner class to handle the events.
- Given a class that extends `JPanel` and includes one or more components that generate events, add anonymous inner classes to handle the events.
- Given a class that extends `JPanel` and includes one or more components that generate events, add event listeners to handle keyboard and focus events.
- Write statements that display dialog boxes using the `JOptionPane` class.
- Given a class that extends `JPanel` and includes one or more text field components, add data validation to the event listeners for the panel.
- Given the requirements for an application that uses the event handling and data validation techniques presented in this chapter, write the code to implement the application.

#### Knowledge

- Distinguish among an event, an event source, an event object, and an event listener.
- Explain the difference between semantic events and low-level events.
- List four options for structuring the classes that implement the listener interface for an event.
- List two options for structuring the classes to handle multiple event sources for a particular event.
- Explain the difference between an inner class and an anonymous inner class.
- Describe at least two situations in which you would want to listen to low-level events.
- Explain the benefit of using adapter classes rather than implementing listener interfaces.

## Summary

---

- An *event* is an object that's generated by user actions or by system events. An *event listener* is an object that implements a listener interface.
- A *semantic event* is an event that's related to a specific component like clicking on a button. In contrast, a *low-level event* is a less specific event like clicking the mouse.
- To handle an event, you must implement the appropriate listener interface. Then, you must add an object created from the listener class to the appropriate component by using the `addEventListener` method, and you must code the methods of the listener interface.
- The class that creates the component that generates events can also serve as the listener for those events. In that case, you specify *this* in the `addEventListener` method.
- You can also use separate classes to listen for events. The benefit of this is that it separates the code that controls the appearance of the user interface from the code that manages the application's behavior.
- *Inner classes* are often used as event listeners because they can easily access the fields and methods of the class that contains them.
- *Anonymous inner classes* are sometimes used as event listeners because they let you easily create classes that are instantiated only once.
- To make it easier to code the listener interfaces for low-level events, the Java API includes *adapter classes* that contain empty methods for all of the methods in the listener interface.

## Terms

---

event	inner class
event source	containing class
event object	anonymous inner class
event listener	anonymous class
register an event listener	focus event
semantic event	keyboard event
low-level event	adapter class

### **Exercise 17-1     Implement an event listener as a separate class**

---

1. Open the FutureValueApp file in the c:\java1.6\ch17\FutureValue directory, and save it in the c:\java1.6\ch17\FutureValueSeparateClass directory.
2. Review the code for this application to see that the ActionListener for the Calculate and Exit buttons are implemented by the FutureValuePanel class.
3. Run the application to refresh your memory on how it works.
4. Modify the application so that the ActionListener for the Calculate and Exit buttons is implemented in a separate class rather than by the FutureValuePanel class.
5. Compile the application, and test it to be sure it still works correctly.

### **Exercise 17-2     Implement an event listener as an inner class**

---

1. Open the FutureValueApp file in the c:\java1.6\ch17\FutureValue directory, and save it in the c:\java1.6\ch17\FutureValueInnerClass directory.
2. Modify the file so that the ActionListener for the Calculate and Exit buttons is implemented as an inner class within the FutureValuePanel class.
3. Compile the application, and test it to be sure it works correctly.

### **Exercise 17-3     Implement separate event listeners**

---

1. Open the FutureValueApp file in the c:\java1.6\ch17\FutureValue directory, and save it in the c:\java1.6\ch17\FutureValueSeparateListeners directory.
2. Modify the file so that the ActionListener listeners for the Calculate and Exit buttons are implemented as separate inner classes within the FutureValuePanel class.
3. Compile the application, and test it to be sure it works correctly.

### **Exercise 17-4     Implement the event listeners as anonymous inner classes**

---

1. Open the FutureValueApp file in the c:\java1.6\ch17\FutureValue directory, and save it in the c:\java1.6\ch17\FutureValueAnonymousClasses directory.
2. Modify the file so that the ActionListener listeners for the Calculate and Exit buttons are implemented as anonymous inner classes in the statements that call the addActionListener method for each button.
3. Compile the application, and test it to be sure it works correctly.

### **Exercise 17-5      Add low-level events to the Future Value application**

---

1. Open the FutureValueApp file in the c:\java1.6\ch17\FutureValue directory.
2. Add low-level event listeners to this application so (1) the user is prevented from entering non-numeric characters into the text fields, and (2) the characters in each text field are selected automatically when the text field receives the focus.
3. Compile the application, and test it to be sure it works correctly.

### **Exercise 17-6      Add data validation to the Future Value application**

---

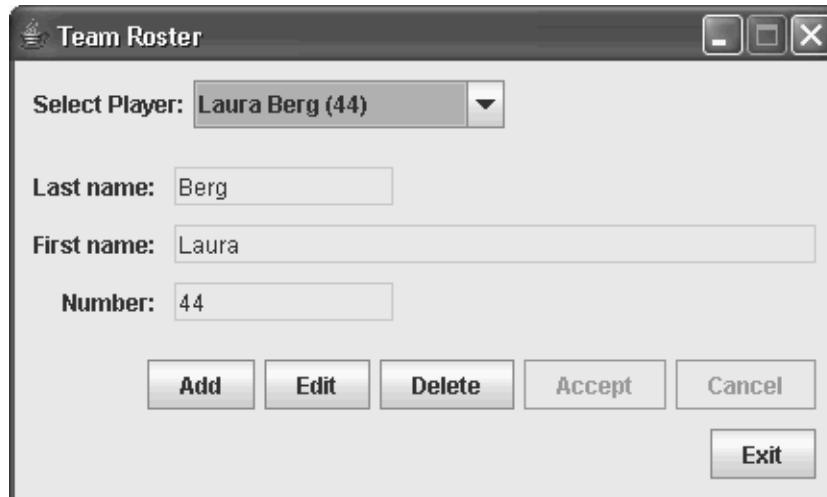
1. Open the FutureValueApp file in the c:\java1.6\ch17\FutureValue directory.
2. Add data validation to the Future Value application so the user is prevented from omitting data or entering non-numeric data in the text fields. Use the SwingValidator class in the c:\java1.6\ch17\FutureValue directory to handle the validation. The user must enter data into all three of the text fields. In addition, the monthly investment and interest rate text fields must be valid double values, and the number of years text field must be an integer.
3. Compile the application, and test it by entering invalid data.



---

## Project 17-1: Maintain a team roster

---



### Operation

- This application maintains a roster for a sports team. Each player has a last name, a first name, and a number.
- When the user selects a player from the combo box, the application lists the player's name and number in the text fields. The user can then click the Edit or Delete button to edit or delete the player.
- If the user clicks the Edit button, the text fields are enabled, the Add, Edit, and Delete buttons are disabled, and the Accept and Cancel buttons are enabled. If the user then clicks the Accept button, any changes the user made to the player's data are saved. If the user clicks Cancel, any changes are discarded.
- If the user clicks the Delete button, the player is immediately removed.
- If the user clicks the Add button, the text fields are cleared and enabled, the Add, Edit, and Delete buttons are disabled, and the Accept and Cancel buttons are enabled. The user can then enter data for a new player and click the Accept button to add the player. If the user clicks the Cancel button, the player is not added.

### Specifications

- Create a Player class to represent a player. You can provide any public fields and methods for this class that you think are appropriate.
- To simplify this project, create a class named TeamIO that simulates the presence of a file. This class should have a static getTeam method that returns an array list of players. The array list should be populated with a roster that's hard coded into the class, so no actual file I/O is performed. The class should also have a static saveTeam method that accepts an array list of Player objects and displays the list on the console. This will help you verify that your program is working.
- When the application starts, it should call the getTeam method to get the team roster. Then, each time the user adds, changes, or deletes a player, the application should call the saveTeam method to save the changes.

- Include data validation that requires entries in all three text fields.
- Create a class named `AutoSelect` that implements the `FocusListener` interface. Use the `focusGained` method of this class to select the text in any of the text fields when it receives the focus.
- Create a class named `IntFilter` that implements the `KeyListener` interface. Use the `keyTyped` method of this class to restrict the input for the player number to the numbers 0 to 9.

### **Enhancements**

- Validate the number field so that no two players can have the same number.
- Implement the `TeamIO` class so that the data is retrieved from a file and changes are saved to the file.
- Require the students to use a factory pattern to obtain the I/O class that gets and saves team data.

---

## Project 17-2: Validate user entries

---



### Operation

- This application accepts user entries and validates them according to the specifications below. If the data is valid, a dialog box with the message “Data accepted” is displayed and the text fields are cleared. If an entry isn’t valid, a dialog box is displayed with an appropriate error message and the focus is moved to the text field with the invalid data.

### Specifications

- Create a class named `Validator` that holds a collection of validation requirements for one or more text fields. The constructors and methods for the `Validator` class follow:

Constructor	Description
<code>Validator(JPanel panel)</code>	Creates a <code>Validator</code> object. The <code>JPanel</code> is used to position any error messages that might be displayed when data is validated.
Method	Description
<code>void addRequiredField(JTextField t, String name)</code>	Adds a required text field.
<code>void addIntegerField(JTextField t, String name)</code>	Adds a text field into which the user must enter a valid integer.
<code>void addDoubleField(JTextField t, String name)</code>	Adds a text field into which the user must enter a valid double value.
<code>void addIntMinField(JTextField t, int min, String name)</code>	Adds a text field into which the user must enter an integer value that is greater than or equal to the specified value. The <code>addInteger</code> method must be called for the text field prior to calling this method.
<code>void addIntMaxField(JTextField t, int max, String name)</code>	Adds a text field into which the user must enter a value that is less than or equal to the specified value. The <code>addInteger</code> method must be called for the text field prior to calling this method.

Method	Description
<code>boolean validate()</code>	Performs the validations that have been created by the various add methods, in the order those methods were called. If any text field fails validation, a message box is displayed to indicate the error, focus is moved to the field with the invalid data, and the validate method returns false. Additional fields are not validated.

- To use the Validator class, you first call the methods that add validation requirements to the Validator object. The Validator object maintains these requirements in an array list. Then, you call the validate method to validate the text fields against the requirements that have been established.
- Create a class with a frame that includes three text fields: name, which is a required field; age, which must be an integer from 0 to 120; and sales, which must be a valid double value.
- The frame must also define a button labeled “OK” that performs the validation. If the data is validated successfully, a dialog box with the message “Data accepted” should be displayed, and the contents of the five text fields should be cleared.

### Hints

- Don’t hesitate to define one or more private classes or interfaces nested within the Validator class to simplify the solution.
- One benefit of defining the Validator class this way is so that you can place the code that sets up the validation for each text field along with the code that creates the fields. Thus, you should place the code that calls the various add methods of the Validator class near the code that sets up the panel layout, not in the code that handles the action event. The event handler code should simply call the validate method to validate the data.

### Enhancements

- Add additional validation options, such as min and max for double values or a range checking validation that combines the min and max functions.
- Modify the Validator class so that instead of displaying error messages in a dialog box, it displays error messages in labels on the panel. The add methods will have to be modified so they accept a JLabel parameter, and the validate method will have to be modified so that instead of displaying a dialog box, it places the error message in the specified label. In addition, the validate method should continue validating fields even after it encounters an invalid field. However, focus should be given to the first field in error.
- Modify the Product Maintenance application that’s presented in the chapter (figures 17-17 through 17-19) to use the Validator class.

## Chapter 18

# How to develop applets

### Objectives

---

#### Applied

- Given a working Swing application, convert the application to an applet.
- Write a simple HTML page that displays an applet.
- Use the Applet Viewer to test an applet.
- Use the Java Plug-in HTML Converter to convert an HTML page that includes an APPLET tag to a form that can be run by Internet Explorer or Netscape browsers.
- Run an applet in a browser window by opening an HTML page that includes the applet.
- Use the jar command to create an archive file that contains an applet, and modify the HTML file that displays the applet so the applet is run from the jar file.
- Given the specifications for a program that uses the Swing components presented in this book, implement the program as an applet.

#### Knowledge

- Explain the difference between a Java applet and an application.
- Describe what happens if a user tries to run an applet without having the appropriate Java Plug-in installed.
- Describe the operations applets are not allowed to do for security reasons.
- List the four methods defined by the Applet class and explain when each of them is called.
- Explain how to modify a Swing application so that it runs as an applet.
- List three attributes that are commonly used with the APPLET tag.
- Explain the purpose of the Applet Viewer.
- Explain the purpose of the Java Plug-in HTML Converter.
- Explain the purpose of the Java Console window and describe how to display it with version 5.0 or later of the JDK.
- Explain the benefits of using jar files to deploy applets.

## Summary

---

- An *applet* is a special type of application that's stored on a web server and runs within a web browser on a client machine.
- To view an applet, the client computer must have the appropriate version of the *Java Runtime Environment (JRE)* and the *Java Plug-in* installed. The Java Plug-in is part of the JRE.
- Since applets are downloaded from remote servers and run on client machines, they have stricter security restrictions than applications. To get around these restrictions, it's possible to create a *signed applet*.
- You can use the `JApplet` class to create an applet that can use Swing components and all of the current features of Java.
- You can use the Applet Viewer to test an applet outside of its HTML page.
- You can use the *Hypertext Markup Language (HTML)* to create a web page. Within the HTML file, you use *tags* to define the elements of the page. And within some tags, you define *attributes* that provide additional information.
- To add an applet to a web page, you include an `APPLET` tag with `CODE`, `HEIGHT`, and `WIDTH` attributes.
- Before you deploy an applet, you should run the Java Plug-in HTML Converter to convert the HTML page so it works with the appropriate browsers and operating systems.
- To test an applet and its HTML page, you view the HTML page in a web browser. Then, you can use the Java Console to view the debugging information.
- You can use a *Java Archive file (JAR file)* to store the files required by an applet in a compressed format. Then, you can modify the HTML page for the applet so it uses this JAR file.

## Terms

---

applet  
Java Plug-in  
signed applet  
Hypertext Markup Language (HTML)  
HTML tag  
attribute  
Java Archive (JAR) file

## Exercise 18-1    Develop a Payment applet

---

In this exercise, you'll modify the Payment application you saw in chapter 16 so that it can be run as an applet.

1. Open the PaymentApp class in the `c:\java1.6\ch18\Payment` directory and review its code.
2. Open a new file, and cut and paste the code for the PaymentPanel class from the PaymentApp file to the new file. Give this class public access, add any required import statements, save the file as PaymentPanel, and compile it.
3. Delete the import statements that aren't needed from the PaymentApp file, then compile the file and run the application to see that it still works.
4. Create a class named PaymentApplet that displays the payment panel. When you're done, compile the applet class to make sure that it compiles cleanly.
5. Open the payment.html file and review the starting code. This file contains all the tags needed for the Payment applet except the APPLET tag. Add an APPLET tag that displays the PaymentApplet at 300 by 300 pixels, and then save the file.
6. Use the Applet Viewer to view and test the applet. When you're done, click the Exit button to see that an exception is thrown indicating that the applet doesn't have the proper security clearance to access the current thread.
7. Modify the PaymentPanel class so it doesn't include an Exit button, and compile this class. Then, run the PaymentApp class and test it. This application won't display an Exit button anymore, but you can still close it by clicking the Close button in the upper right corner.
8. Use the Applet Viewer to run and test the PaymentApplet class. This time, you'll have to exit from the applet by closing the browser window. Now, you have an application (PaymentApp) and an applet (PaymentApplet) that both use the same panel class.
9. Run the HTML Converter to convert the payment.html page. Then, open the payment.html page that's in the Payment directory and review the new code. Also, open the payment.html page that's in the Payment\_BAK directory to see that it contains the original code.
10. Use your web browser to test the HTML page and the applet. Since the JRE and Java Plug-in were automatically installed when you installed the JDK, the applet should work properly.
11. Modify the actionPerformed method of the PaymentPanel class so that when you click the Accept button, a message is printed to the console indicating that the payment was accepted. Then, compile the class and run the applet from the browser again. This time, display the Java Console and view the statement that's printed to the console when you click the Accept button.

## **Exercise 18-2     Store the Payment applet in a JAR file**

---

In this exercise, you'll create a JAR file that contains the class files for the Payment applet that you created for exercise 18-1.

1. Use the JAR tool to create a JAR file named `payment.jar` that contains only the class files needed by the Payment applet.
2. Replace the `payment.html` file that's in the Payment directory with the `payment.html` file that's in the `Payment_BAK` directory. Modify the `payment.html` page so it uses the JAR file you created in step 1.
3. Use the HTML Converter to convert the HTML page. Then, use your web browser to display the HTML page to see that it still works with the JAR file.
4. If you have access to a web server, use an FTP program to upload the `payment.html` and `payment.jar` files to the web server. Then, run the Payment applet from the web server. If you have access to systems that don't have the current version of Java installed, test this applet on these systems to see what happens.



---

## Project 18-1: Simulate a vending machine

---



### Operation

- This project is implemented as an applet that simulates a vending machine that dispenses soft drinks. The user interface has the following controls:
- Quarter button: Deposits \$0.25.
- Dollar button: Deposits \$1.00.
- A label that indicates the current credit.
- Refund button: Refunds the customer's credit.
- Slot buttons 1-6: These buttons display the name of the product currently loaded in one of the vending machine's six internal slots. The example above shows that the vending machine is loaded with Pepsi, Diet Pepsi, Mountain Dew, Dr. Pepper, Root Beer, and Water.
- When the applet is first started, the six slot buttons are disabled. These buttons are enabled only when the user has a credit of at least \$1.00 (the cost of one item).

- When the user clicks one of the slot buttons, the program displays a dialog box that tells the user to enjoy the beverage. (The message should indicate the specific beverage selected. For example, “Enjoy your Diet Pepsi.”)
- If the user clicks Refund, a dialog box should appear that asks the user to take his or her change. For example, “Please take your change of \$1.25.”

### **Specifications**

- You may design any classes you wish to use for this application.
- Each slot in the vending machine can hold up to 10 bottles. The application should keep track of how many bottles are available in each slot and display a dialog box with the message “Out of Stock” if the user selects a beverage that’s out of stock.
- Create a web page to display the application.

### **Hint**

- To set the size of a button, use the `setPreferredSize` method. It accepts a `Dimension` object as a parameter.

### **Enhancements**

- Load the beverages from a file.
- Add an indicator (such as an asterisk) to the “slot” buttons that shows when the beverages are sold out. Alternatively, disable a button when the beverage is sold out.

## Chapter 19

# How to work with text and binary files

### Objectives

---

#### Applied

- Write code that uses the File class to get information about a file or directory.
- Write code that reads and writes data to a text file using buffered readers and writers.
- Use the string handling features that were presented in chapter 12 to process text read from a delimited text file.
- Write code that reads and writes data to a binary file using primitive data types and fixed-length strings.
- Write code that reads and writes random-access files.
- Given the specifications for an application that stores its data in a text or binary file, implement the program using the file handling features presented in this chapter.

#### Knowledge

- Explain the differences between a text file and a binary file.
- Explain the concept of layering and how it is used to create filtered streams that can read or write files.
- Explain how a buffer for an output stream works and how it improves the performance of an I/O operation.
- Name and describe the three common types of I/O exceptions.
- Describe the functions provided by the BufferedWriter, PrintWriter, and FileWriter classes when writing data to a text file.
- Describe the functions provided by the BufferedReader and FileReader classes when reading data from a text file.
- Describe the functions provided by the DataOutputStream, BufferedOutputStream, and FileOutputStream classes when writing data to a binary file.
- Describe the functions provided by the DataInputStream, BufferedInputStream, and FileInputStream classes when reading data from a binary file.
- List two ways that strings can be stored in a binary file and describe the difference between the techniques.
- Explain the differences in the use of random-access and sequential-access files.

## Summary

---

- A *text file* stores data as characters. A *binary file* stores data in a binary format.
- In a *delimited text file*, *delimiters* are used to separate the *fields* and *records* of the file.
- You use *character streams* to read and write text files and *binary streams* to read and write binary files. To get the functionality you need, you can *layer* two or more streams.
- A *buffer* is a block of memory that is used to store the data in a stream before it is written to or after it is read from an I/O device. When an output buffer is full, its data is *flushed* to the I/O device.
- When you work with I/O operations, you'll need to catch or throw three types of checked exceptions: `IOException`, `FileNotFoundException`, and `EOFException`.
- To identify a file when you create a `File` object, you can use an *absolute path name* or a *relative path name*. To identify a file on a remote computer, you can use the *Universal Naming Convention (UNC)*.
- The `File` class provides many methods that you can use to check whether a file or directory exists, to get information about a `File` object, and to create or delete directories and files.
- You can use the classes in the `Writer` and `Reader` hierarchies to work with a text file. You can use the classes in the `OutputStream` and `InputStream` hierarchies to work with a binary file. You can also use the methods of the `DataOutput` and `DataInput` interfaces to work with binary files.
- You can use the `RandomAccessFile` class to access a binary file randomly rather than sequentially. When you use a *random-access file*, you can position a *pointer* to any location in the file.
- When you work with random-access files, you store string values as *fixed-length strings*. That way, the files have the same number of bytes for each field within each record.

## Terms

---

absolute path name	input stream	column
relative path name	character stream	record
Universal Naming Convention (UNC)	binary stream	row
input file	layer two or more streams	Universal Text Format (UTF)
output file	buffered stream	sequential-access file
I/O operations	buffer	sequential file
file I/O	flush the buffer	random-access file
text file	autoflush feature	pointer
binary file	delimited text file	cursor
stream	delimiter	metadata
output stream	field	fixed-length string

## Exercise 19-1 Work with a text file

---

In this exercise, you'll create an application that maintains the name of the class that's used to create the ProductDAO object for the Product Maintenance application of chapter 8 in a text file. Then, you'll modify the Product Maintenance application so it uses the object specified in the text file.

### Create the DAO Maintenance application

1. Open the DAOFile and DAOMaintApp classes in the `c:\java1.6\ch19\ProductMaintenance` directory.
2. Add code to the DAOFile class so that it can read and write a string that contains the name of a ProductDAO class to a text file named `dao.txt`. To do that, you'll need to add code to the constructor to initialize the File object, and you'll need to add code to the `getDAOName` and `setDAOName` methods so they read and write to the file.
3. Display the DAOMaintApp class, and add the code needed to create a DAOFile object and to get and set the name of the ProductDAO object in the file that the DAOFile object refers to. Test this application by changing the name of the ProductDAO class to `ProductTextFile`. The output should look like this:

```
Welcome to the DAO Maintenance application

The current ProductDAO class is: ProductRandomFile

Do you want to change this? (y/n): y

Enter the name of a valid ProductDAO class: ProductTextFile

The current ProductDAO class is: ProductTextFile

Do you want to change this? (y/n): n
```

4. To be sure that the `dao.txt` file was changed correctly, open it in a text editor.

### Modify the Product Maintenance application

5. Open the DAOFactory class and modify it so it uses the DAOFile class to read the string in the `dao.txt` file and return the appropriate DAOFactory object. It should provide for `ProductTextFile` and `ProductRandomFile` objects.
6. Open the ProductMaintApp class and test it to make sure it works correctly. If the name of the class in the `dao.txt` file is anything other than `ProductTextFile` or `ProductRandomFile`, this class will throw an exception.
7. Modify the ProductMaintApp class so it prevents the exception that's thrown if an invalid ProductDAO class is specified.

## Exercise 19-2 Work with a binary file

---

In this exercise, you'll enhance the Product Maintenance application so it can use a binary file.

1. Open the ProductTextFile class in the c:\java1.6\ ProductMaintenance directory. Change the class name to ProductBinaryFile, and save the class with this name.
2. Modify this class so it uses a binary file named products.dat. Be sure to store the product code and description as UTF characters.
3. Modify the DAOFactory class so it provides for the ProductBinaryFile class.
4. Run the DAOMaintApp application to change the ProductDAO class to ProductBinaryFile. Then, run the ProductMaintApp application to see if it works with the binary file. Make sure to leave at least three product records in the binary file.
5. Use a text editor to open the products.dat file. Note that it's easier to read the data that's stored in the text file.

## Exercise 19-3 Improve the exception handling

---

In this exercise, you'll improve the way that exceptions are handled by the ProductMaintApp and ProductTextFile classes.

1. Open the ProductMaintApp class and modify it so it displays an error message and exits the application if the getProducts method returns a null.
2. Open the ProductTextFile class and comment out the line in the catch clause of the getProducts method that prints the stack trace to the console. Then, add a statement like this that throws an IOException to the beginning of the getProducts method:

```
if (true)
    throw new IOException(
        "This is a test of the getProducts method.");
```

3. Test these changes to make sure they're working correctly. To do that, run the DAOMaintApp application to set the name of the ProductDAO class to ProductTextFile. Then, run the ProductMaintApp application and enter the list command. If this works, comment out the statement you added that throws the IOException.
4. Modify the ProductMaintApp class so it responds appropriately if the addProduct or deleteProduct method returns a false value. Test this exception by commenting out the line in the catch clause of the saveProducts method that displays the stack trace and adding a statement near the beginning of this method that throws an IOException. Test these changes.
5. Modify the ProductTextFile class so it writes exceptions to a text file named errorLog.txt instead of printing them to the console. To do that, add a method named printToLogFile that accepts an IOException as an argument. This method should append two records to the log file: one that indicates the date and time the exception occurred and one that contains information about the exception.

6. Modify the `getProducts` and `saveProducts` methods so they call the `printToLogFile` method when an error occurs. Test these changes. When you're sure this works correctly, comment out the statement you added that throws the `IOException` and compile the class again.

### **Exercise 19-4    Enhance the random-access processing**

---

In this exercise, you'll modify the `ProductRandomFile` class so it works more efficiently.

1. Use the `DAOMaintApp` application to specify the `ProductRandomFile` class as the `ProductDAO` class.
2. Open the `ProductRandomFile` class in the `c:\java1.6\ch19\ProductMaintenance` directory and review its code. Note that the `getProducts` method does not use a buffered input stream.
3. Modify the `getProducts` method so it uses a buffered `DataInputStream` to read each product in the `products.ran` file sequentially, creates a `Product` object for each product that isn't marked for deletion, and adds the `Product` object to the `products` array list. Be sure to check that the file exists before you process it.
4. Test this change by running the `ProductMaintApp` application.

### **Exercise 19-5    Update the random-access file**

---

In this exercise, you'll add a method to the `RandomAccessFile` class that can be used to permanently delete the records in the `products.ran` file that have been marked for deletion. Then, you'll write the code to call this method.

1. Open the `ProductRandomFile` and `RandomMaintApp` classes in the `c:\java1.6\ch19\ProductMaintenance` directory.
2. Add a method named `commitDeletions` to the `ProductRandomFile` class. This method should delete all the records from the `products.ran` file that are marked for deletion and return the number of records that were deleted. (If an `IOException` occurs, it should return `-1`.) To do that, you can write all the products in the `products` array list to the file, and you can use the `setLength` method to set the file to the appropriate length. To make this code work, you'll need to close the `RandomAccessFile` object so that the new file length is applied. Also, be sure to reopen the file for random access, and reinitialize the `productCodes` array so it contains only the current products.
3. Display the `RandomMaintApp` class and note that it's similar to the `ProductMaintApp` class but contains only `commit`, `help`, and `exit` commands. Add the code to this class to implement the `commitDeletions` method that's executed when the user enters the `commit` command. If the `commit` operation is successful, it should display the number of records that were deleted. Otherwise, it should display an appropriate error message. Test this class to be sure it works correctly. To do that, you'll need to use the `ProductMaintApp` application to delete one or more records before you run the `RandomMaintApp` application.

---

## Project 19-1: Maintain a list of countries

---

### Console

```
Welcome to the Countries Maintenance application

1 - List countries
2 - Add a country
3 - Exit

Enter menu number: 1

India
Japan
Mexico
Spain
United States

1 - List countries
2 - Add a country
3 - Exit

Enter menu number: 2

Enter country: Thailand

This country has been saved.

1 - List countries
2 - Add a country
3 - Exit

Enter menu number: 1

India
Japan
Mexico
Spain
United States
Thailand

1 - List countries
2 - Add a country
3 - Exit

Enter menu number: 3

Goodbye.

Press any key to continue . . .
```



## Operation

- The application begins by displaying a menu with three choices.
- If the user chooses the first menu item, the application displays a list of countries that are saved in a file.
- If the user chooses the second menu item, the application prompts the user to enter a country and then it writes that country to the file of countries.
- If the user chooses the third menu item, the application displays a goodbye message and exits.

## Specifications

- Create a class named `CountriesTextFile` that contains one method that allows you to read a list of countries from a file and another method that allows you to write a list of countries to a file. For example:

```
public ArrayList<String> getCountries()  
public boolean saveCountries(ArrayList<String> countries)
```
- Store the list of countries in a text file named `countries.txt` in the same directory as the `CountriesTextFile` class. If the `countries.txt` file doesn't exist, the `CountriesTextFile` class should create it. This class should use buffered I/O streams, and it should close all I/O streams when they're no longer needed.
- Create a class named `CountriesApp` that displays the menu and responds to the user's choices.
- Use the `Validator` class presented in chapter 6 or an enhanced version of it to validate the user's entries. A valid integer is required for the menu choice, and a non-empty string is required for the country.

## Enhancements

- Create another class named `CountriesBinaryFile` that can store the list of countries in a binary data file named `countries.dat`. This class should contain a main method that initializes the file for the first time by writing several countries to it. Then, modify the `CountriesApp` class so it uses the `CountriesBinaryFile` class instead of the `CountriesTextFile` class.
- Modify the `CountriesApp` class so it includes a menu choice that allows the user to delete a country from the file.

---

## Project 19-2: Maintain a conversions table

---

### Console

```
Welcome to the Length Converter

1 - Convert a length
2 - Add a type of conversion
3 - Delete a type of conversion
4 - Exit

Enter menu number: 1

1 - Miles to Kilometers: 1.6093
2 - Kilometers to Miles: 0.6214
3 - Inches to Centimeters: 2.54

Enter conversion number: 2

Enter Kilometers: 10
10.0 Kilometers = 6.214 Miles

1 - Convert a length
2 - Add a type of conversion
3 - Delete a type of conversion
4 - Exit

Enter menu number: 2

Enter 'From' unit: Centimeters
Enter 'To' unit: Inches
Enter the conversion ratio: .3937

This entry has been saved.

1 - Convert a length
2 - Add a type of conversion
3 - Delete a type of conversion
4 - Exit

Enter menu number: 1

1 - Miles to Kilometers: 1.6093
2 - Kilometers to Miles: 0.6214
3 - Inches to Centimeters: 2.54
4 - Centimeters to Inches: 0.3937

Enter conversion number: 4

Enter Centimeters: 2.54
2.54 Centimeters = 1 Inches

1 - Convert a length
2 - Add a type of conversion
3 - Delete a type of conversion
4 - Exit

Enter menu number: 4

Goodbye.

Press any key to continue . . .
```

## Operation

- This application begins by displaying a main menu with four items: (1) Convert a length, (2) Add a type of conversion, (3) Delete a type of conversion, and (4) Exit.
- If the user chooses the first main menu item, the application displays a menu of possible conversions. After the user selects a conversion, the application prompts the user to enter a unit of measurement, calculates the conversion, displays the result, and displays the main menu again.
- If the user chooses the second main menu item, the application prompts the user to enter the values for a new conversion, saves this new conversion to a file, and displays a message to the user.
- If the user chooses the third main menu item, the application displays a menu of possible conversions. After the user selects the conversion, the application deletes that conversion from the file, displays a message to the user, and displays the main menu again.
- If the user chooses the fourth main menu item, the application displays a goodbye message and exits.

## Specifications

- Create a class named `Conversion` that can store information about a conversion, including `fromUnit`, `fromValue`, `toUnit`, `toValue`, and `conversionRatio`. This class should also contain the methods that perform the conversion calculations and return the results as a formatted string.
- Create a class named `ConversionsTextFile` that contains one method that reads an array list of `Conversion` objects from a file and another that writes an array list of `Conversion` objects to a file. For example:

```
public ArrayList<Conversion> getConversions()  
public boolean saveConversions(ArrayList<Conversion> typesList)
```
- Store the list of conversions in a text file named `conversion_types.txt` in the same directory as the `ConversionsTextFile` class. If the `conversion_types.txt` file doesn't exist, the `ConversionsTextFile` class should create it. This class should use buffered I/O streams, and it should close all I/O streams when they're no longer needed.
- Create a class named `ConversionsApp` that displays the menus shown in the console output and responds to the user's choices.
- Use the `Validator` class or a variation of it to validate the user's entries. A valid integer is required for a menu choice, non-empty strings are required for the "From" and "To" fields, and a valid double is required for the conversion ratio.

---

## Project 19-3: Maintain customer data (text or binary file)

---

### Console

```
Welcome to the Customer Maintenance application

COMMAND MENU
list    - List all customers
add     - Add a customer
del     - Delete a customer
help   - Show this menu
exit   - Exit this application

Enter a command: list

CUSTOMER LIST
frank46@hotmail.com      Frank      Jones
sarah_smith@yahoo.com   Sarah      Smith

Enter a command: add

Enter customer email address: test@gmail.com
Enter first name: text
Enter last name: test

text test was added to the database.

Enter a command: list

CUSTOMER LIST
frank46@hotmail.com      Frank      Jones
sarah_smith@yahoo.com   Sarah      Smith
test@gmail.com           text      test

Enter a command: del

Enter customer email to delete: test@gmail.com

text test was deleted from the database.

Enter a command: list

CUSTOMER LIST
frank46@hotmail.com      Frank      Jones
sarah_smith@yahoo.com   Sarah      Smith

Enter a command: exit

Bye.

Press any key to continue . . .
```

## Operation

- This application begins by displaying a menu with five choices: list, add, del, help, and exit.
- If the user enters “list”, the application displays the customer data that’s stored in a text file.
- If the user enters “add”, the application prompts the user to enter data for a customer and saves that data to the text file.
- If the user enters “del”, the application prompts the user for an email address and deletes the corresponding customer from the text file.
- If the user enters “help”, the application displays the menu again.
- If the user enters “exit”, the application displays a goodbye message and exits.

## Specifications

- Create a class named `Customer` that stores data for the user’s email address, first name, and last name.
- Create interfaces named `CustomerReader` and `CustomerWriter` that define the methods that will be used to read and write customer data to a customer file. In addition, create an interface named `CustomerConstants` that contains three constants that specify the display size of a customer’s email address (30), first name (15), and last name (15). Then, create an interface named `CustomerDAO` that inherits all three of these interfaces.
- Create a class named `CustomerTextFile` that implements the methods specified by the `CustomerDAO` interface. Store the customer data in a text file named “customers.txt” in the same directory as this class. If the customers.txt file doesn’t exist, this class should create it. This class should use buffered I/O streams, and it should close all I/O streams when they’re no longer needed.
- Create a class named `DAOFactory` that contains a method named `getCustomerDAO`. This method should return an instance of the `CustomerTextFile` class.
- Create a `CustomerMaintApp` class that displays the prompts shown in the console output and accepts the user entries. This class should use the `DAOFactory` class to get a `CustomerDAO` object. Then, it should use the methods of the `CustomerDAO` object to read customer data from the file and to write customer data to the file.
- Use the `Validator` class or a variation of it to validate the user’s entries. Non-empty strings are required for the email address, first name, and last name.
- Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named `StringUtils` with a method that adds the necessary spaces to a string to reach a specified length.

## Enhancements

- Modify the application so it uses a class named `CustomerBinaryFile` to store the data in a binary file named `customer.dat`.
- Modify the application so it uses a class named `CustomerRandomFile` to store the data in a random-access file named `customer.ran`. This class should use the `seek` method to randomly access each customer record whenever that's necessary.
- Add an "update" command that lets the user update an existing customer. This command should prompt the user to enter the email address. Then, it should let the user update the first name or last name for the customer.
- Add a method to the `Validator` class that uses string parsing techniques to validate the email address. At the least, you can check to make sure that this string contains some text, followed by an `@` sign, followed by some more text, followed by a period, followed by some more text. For example, "x@x.x" would be valid while "xxx" or "x@x" would not.
- **EXTRA CREDIT:** Create a GUI for this application instead of the console. Use the Product Maintenance program presented in chapter 17 as a model.

## Chapter 20

# How work with XML

### Objectives

---

#### Applied

- Given a listing of an XML document, identify the elements and content it contains.
- Use a web browser or XML editor to view or edit XML files.
- Use the classes of the StAX API to read and write the data that's stored in an XML file.

#### Knowledge

- Describe the major differences between XML and HTML.
- Explain how elements are structured in an XML document.
- Explain the difference between a child element and an attribute, and describe the relative advantages of each.
- Describe the use of a DTD.
- Explain why using StAX is generally preferable to using either SAX or DOM.

### Summary

---

- *XML* provides a standard way to structure data by using *tags* that identify data items.
- An *element* begins with a *start tag* and ends with an *end tag*. An element can contain data in the form of *content* that appears between the tags. It can also contain *child elements*.
- An *attribute* consists of a name and value that appear within an element's start tag.
- A *DTD (Documentation Type Definition)* is a *schema* that defines the structure of an XML document. This schema can be enforced when a document is read or written.
- You can use a web browser to view XML data, and you can use any text editor to edit an XML file, but it's helpful to use a text editor that's designed for working with XML.
- *DOM (the Document Object Model)* is an API that can be used to build a DOM tree, work with the nodes of a tree, read an XML document from a file, and write an XML document to a file.
- *SAX (the Simple API for XML)* can be used to read an XML document.
- *StAX (the Streaming API for XML)* is appropriate for reading or writing XML documents of all sizes.

## Terms

---

Extensible Markup Language (XML)  
XML document  
tag  
XML declaration  
element  
start tag  
end tag  
content  
child element  
parent element  
root element  
attribute  
schema  
schema language  
Document Type Definition (DTD)  
Simple API for XML (SAX)  
Document Object Model (DOM)  
Streaming API for XML (StAX)



## Exercise 20-1 Work with an XML file

In this exercise, you'll write code that works with an XML document that's stored in a file. When you complete this exercise, the console output should look like this:

```
Products list:
java      Murach's Beginning Java          $49.50
jsps      Murach's Java Servlets and JSP        $49.50

XML Tester has been added to the XML document.

Products list:
java      Murach's Beginning Java          $49.50
jsps      Murach's Java Servlets and JSP        $49.50
test      XML Tester                          $77.77

XML Tester has been deleted from the XML document.

Products list:
java      Murach's Beginning Java          $49.50
jsps      Murach's Java Servlets and JSP        $49.50
```

1. Use a web browser to view the products.xml file in the ch20\XMLTester directory. Then, collapse and expand some of the elements. If you can't do that, you may need to allow blocked content. With the Internet Explorer, for example, you can click on the information bar and select Allow Blocked Content from the shortcut menu. When you're through experimenting, close the browser, which will reset your content blocker.
2. Open the XMLTesterApp class that's stored in the ch20\XMLTester directory. Run this application to see how it works. At this point, it prints three messages to the console, but it doesn't work with the XML file.
3. Add code to the readProducts method that reads an XML document from the products.xml file and stores it in an array list. Be sure to catch any exceptions that may be thrown. Compile this class to make sure it doesn't contain any compile-time errors. Then, run the class. At this point, the application should print three identical product lists, and those lists should match the data that's stored in the products.xml file.
4. Add code to the writeProducts method that writes the XML document to the products.xml file. Then, remove the comments from the code in the main method that adds and removes a product from the list. Finally, compile and test the application again. When you do, it should display the products list, write the "XML Tester" product to the products.xml file, display the products list again, remove the "XML Tester" product from the file, and display the products list a third time.

## Exercise 20-2 Use an XML file with the Product Maintenance application

---

In this exercise, you'll modify the Product Maintenance application of chapter 8 so it uses an XML file for getting and saving the product records.

1. Open the `DAOFactory` class that's stored in the `ch20\ProductMaintenance` directory. Here, the `getProductDAO` method has already been changed so this application will use the methods in the `ProductXMLFile` class to work with the XML file.
2. Open the `ProductXMLFile` class that's in this directory. This is the code that's shown in figure 20-11, but the `getProducts` and `saveProducts` methods aren't complete. If you look at the constructor for this class, you can see that it creates a `File` object that represents the file named `products.xml`. If you're interested, you can use your web browser to review the data in this file.
3. Open the `ProductMaintApp` class that's in this directory. Its code is the same as it was in chapter 8. Then, run this class to remind yourself how it works. Just enter the exit command when prompted, though, because the other commands won't work until you complete two of the methods in the `ProductsXMLFile` class.
4. Complete the `getProducts` method in the `ProductsXMLFile` class so it returns the `Product` objects that are stored in the XML file. If you've done exercise 20-1, you can do that by copying some of the code from that application's `readProducts` method. Then, test the `getProducts` method to make sure it works correctly by using the list command of the Product Maintenance application.
5. Complete the `saveProducts` method so it writes the `Product` objects that are passed to it to the XML file. If you've done exercise 20-1, you can do that by copying some of the code from that application's `writeProducts` method. Then, test the `saveProducts` method to make sure it works correctly by using the add and delete commands of the Product Maintenance application.
6. To verify that the data is being saved correctly, you can view the XML file in a web browser.

---

## Project 20-1: List artists and albums

---

### Console

```
Artist and Album Listing

Artists
-----
The Beatles
Elvis Presley
John Prine

Albums
-----
Rubber Soul
Revolver
Sgt. Pepper's Lonely Hearts Club Band
The White Album
Elvis at Sun
Elvis 30 #1 Hits
John Prine
Sweet Revenge

Artists and Albums
-----
The Beatles
    Rubber Soul
    Revolver
    Sgt. Pepper's Lonely Hearts Club Band
    The White Album
Elvis Presley
    Elvis at Sun
    Elvis 30 #1 Hits
John Prine
    John Prine
    Sweet Revenge

Press any key to continue . . .
```

### Operation

- This application reads an XML file and displays a list of artists, albums, and albums by artist.

### Specifications

- Create a class named `MusicArtistsApp` that reads an XML file named `music_artists.xml` and displays a list like that shown in the console output above.
- The `music_artists.xml` is provided for you and contains the data shown above.

---

## Project 20-2: Maintain customer data (XML file)

---

### Console

```
Welcome to the Customer Maintenance application

COMMAND MENU
list    - List all customers
add     - Add a customer
del     - Delete a customer
help   - Show this menu
exit   - Exit this application

Enter a command: list

CUSTOMER LIST
frank46@hotmail.com      Frank      Jones
sarah_smith@yahoo.com   Sarah      Smith

Enter a command: add

Enter customer email address: xml_test@gmail.com
Enter first name: XML
Enter last name: Test

XML Test was added to the database.

Enter a command: list

CUSTOMER LIST
frank44@hotmail.com      Frank      Jones
sarah_smith@yahoo.com   Sarah      Smith
xml_test@gmail.com      XML        Test

Enter a command: del

Enter customer email to delete: xml_test@gmail.com

XML Test was deleted from the database.

Enter a command: list

CUSTOMER LIST
frank44@hotmail.com      Frank      Jones
sarah_smith@yahoo.com   Sarah      Smith

Enter a command: exit

Bye.

Press any key to continue . . .
```

## Operation

- This application begins by displaying a menu with five choices: list, add, del, help, and exit.
- If the user enters “list”, the application displays the customer data that’s stored in an XML file.
- If the user enters “add”, the application prompts the user to enter data for a customer and saves that data to the XML file.
- If the user enters “del”, the application prompts the user for an email address and deletes the corresponding customer from the XML file.
- If the user enters “help”, the application displays the menu again.
- If the user enters “exit”, the application displays a goodbye message and exits.

## Specifications

- Create a class named Customer that stores data for the user’s email address, first name, and last name.
- Use a text editor to create an XML file named customers.xml in the same directory as the Customer class. This file should contain valid XML tags for at least one customer. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Customers>
  <Customer Email="frank44@hotmail.com">
    <FirstName>Frank</FirstName>
    <LastName>Jones</LastName>
  </Customer>
</Customers>
```
- Create a class named CustomerXMLFile that reads the customers.xml file when it’s instantiated. This class should include a public method that return an array list of Customer objects created from the data in the file, a public method that returns a Customer object for a specified email address, and public methods for adding and deleting records. Include any additional private methods that you need to perform these functions.
- Create a CustomerMaintApp class that works as shown in the console output. This class should use the Customer and CustomerXMLFile classes to work with Customer data.
- Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named StringUtils that has a method that adds the necessary spaces to a string to reach a specified length.
- Use the Validator class or a variation of it to validate the user’s entries. Non-empty strings are required for the email address, first name, and last name.

## Enhancements

- Add an “update” command that lets the user update an existing customer. This command should prompt the user to enter the email address for a customer. Then, it should let the user update the first name and last name for the customer.
- Add a method to the Validator class that uses string parsing techniques to validate the email address. At the least, you can check to make sure that this string contains some text, followed by an @ sign, followed by some more text, followed by a period, followed by some more text. For example, “x@x.x” would be valid while “xxx” or “x@x” would not.
- Use an interface to eliminate any direct calls to the CustomerXMLFile class from the CustomerMaintApp class. To do that, you can use CustomerReader, CustomerWriter, CustomerConstants, and CustomerDAO interfaces as well as a DAOFactory class as described in project 19-3. Then, you can modify the CustomerXMLFile and CustomerMaintApp classes so they use these interfaces and class.

## Chapter 21

# How to use JDBC to work with databases

### Objectives

---

#### Applied

- Given the arrangement of rows and columns in a table, write SQL statements to retrieve, insert, update, and delete rows from the table.
- Given an existing database and a database server that supports ODBC, use the ODBC Data Source Administrator to register the ODBC data source.
- Given the URL, user name, and password required to connect to an ODBC data source, write the code necessary to create a Connection object that connects to the database.
- Write code that executes a SELECT statement and processes the results using a forward-only or scrollable result set.
- Write code that inserts, updates, or deletes rows in a table.
- Given code that executes a SQL statement, modify the code so it uses a prepared statement and parameters.
- Given a result set, write code that determines the name and data type of each of its columns.
- Given the specifications for an application that stores its data in a relational database, implement the program using the JDBC features presented in this chapter.

#### Knowledge

- Explain how data is organized in a relational database.
- Explain the difference between primary and foreign keys.
- Explain what the SELECT, INSERT, DELETE, and UPDATE statements do.
- Explain what a result set is.
- Explain what a join is, and describe the difference between an inner join and an outer join.
- Identify the four types of Java database drivers and describe the benefits and drawbacks of each driver type.
- Describe the use of automatic driver loading and iterable SQL exceptions.
- Explain the difference between a forward-only, read-only result set and a scrollable, updateable result set.
- Explain what metadata is and how it can be used in database applications.

## Summary

---

- A *relational database* uses tables to store and manipulate data. Each table contains one or more *rows*, or *records*, while each row contains one or more *columns*, or *fields*.
- A *primary key* is used to identify each row in a table. A *foreign key* is a key in one table that is used to relate rows to another table.
- Each database is managed by a *database management system (DBMS)* that supports the use of the *Structured Query Language (SQL)*. To manipulate the data in a database, you use the SQL `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
- The `SELECT` statement is used to return data from one or more tables in a *result set*. To return data from two or more tables, you *join* the data based on the data in related fields. An *inner join* returns a result set that includes data only if the related fields match.
- With JDBC 4.0, the *database driver* that's used to connect the application to a database is loaded automatically. This is known as *automatic driver loading*. With JDBC 4.0, you can also loop through any exceptions that are nested within the SQL Exception object.
- A Java program can use one of four driver types to access a database. *Type-1* and *type-2 drivers* run on the client's machine, while *type-3* and *type-4 drivers* can run on a server machine.
- You can use JDBC to execute SQL statements that select, add, update, or delete one or more records in a database. You can also control the location of the *cursor* in the result set.
- You can use *prepared statements* to supply parameters to SQL statements.
- You can return a list of the column names and types in a result set by using *metadata*. To do this, you may need to convert SQL data types to Java data types.



## Terms

---

database	join
relational database	inner join
table	equi-join
row	calculated field
record	outer join
column	left outer join
field	right outer join
primary key	action query
database management system (DBMS)	Java Database Connectivity (JDBC)
relational database management system (RDBMS)	database driver
foreign key	JDBC-ODBC bridge driver
one-to-many relationship	Open Database Connectivity (ODBC)
one-to-one relationship	native protocol partly Java driver
many-to-many relationship	net protocol all Java driver
default value	native protocol all Java driver
Structured Query Language (SQL)	automatic driver loading
Data Definition Language (DDL)	BLOB objects (Binary Large Objects)
Data Manipulation Language (DML)	CLOB objects (Character Large Objects)
query	prepared statement
result set	metadata
result table	column name
current row pointer	column label
cursor	

## Exercise 21-1      Work with JDBC

---

In this exercise, you'll install an ODBC driver for the MurachDB database, and you'll write JDBC code that works with the MurachDB database.

1. Install an ODBC data source named MurachDB for the MurachDB database that's in the ch21\Databases directory as described in figure 21-8. This database is in Microsoft Access 2000 format. If the ODBC driver for Microsoft Access on your system doesn't support the Access 2000 format, use the database named MurachDB97 instead, but still use MurachDB as the name for the data source.
2. Open the DBTesterApp class that's in the ch21\DBTester directory. Review the code and then run the application. It should print all of the records in the Products table to the console. Then, it should print several blank products to the console.
3. Write the code for the printFirstProduct method. Use column names to retrieve the column values. Then, compile the class and run the application. You can tell if this method is working correctly if it prints the first product in the list of products that's printed by the printProducts method.
4. Write the code for the printLastProduct method. Note that to move to the last product in the result set, you'll need to use a scrollable result set. Compile the class and then test the application.
5. Write the code for the printProductByCode method. Use a prepared statement to create the result set, and use indexes to retrieve the column values. Compile the class and test the application.
6. Write the code for the insertProduct method. This method should begin by checking if a product with the specified product code exists in the database. If so, this method should display an error message. Otherwise, it should add the product to the database and print the product that was added to the console.
7. Compile the class and test the insertProduct method. To do that, you will need to run the application twice. The first time, the product should be added to the database. The second time, the product should appear in the list of products, but then an error message should be displayed indicating that the product already exists.
8. Repeat steps 6 and 7 for the deleteProduct method. You can use this method to delete the product that was added by the insertProduct method. Compile the class and test the method.

## **Exercise 21-2     Use an Access database with the Product Maintenance application**

---

This exercise works with a version of the Product Maintenance application of chapter 8 that uses an Access database for the product data. To work with that data, this application uses the ProductDB class that's presented in figure 21-16.

### **Install an ODBC data source and review the application code**

1. Make sure an ODBC data source named MurachDB is installed for the MurachDB database that's in the ch21\Databases directory as shown in figure 21-8 and described in step 1 of exercise 21-1.
2. Open the DAOFactory class that's stored in the ch21\ProductMaintenance directory. Here, the getProductDAO method has been changed so this application will use the methods in the ProductDB class to work with the Access database.
3. Open the ProductDB class. This is the code that's shown in figure 21-16.
4. Open the ProductMaintApp class and review its code. Its code is the same as it was in chapter 8. Then, run this application. It should work the same as it did in chapter 8, but now the data is stored in an Access database.

### **Modify the way the Connection object is used in the ProductDB class**

5. Display the ProductDB class, and note that the Connection object isn't explicitly closed by this class. Then, delete the instance variable for the Connection object and delete the statement in the constructor that calls the connect method to initialize this instance variable.
6. Modify the declaration for the connect method so it returns a Connection object. Then, modify the code for this method so it creates a Connection object and returns it.
7. Modify the getProduct, addProduct, updateProduct, and deleteProduct methods so they call the connect method to return a Connection object, use this connection object, and close it when they're done with it. Then, compile and test the application. It should work the way it did before.

### **Add a new method to the ProductDB class**

8. Add a new method to the ProductDB class called listProducts that lists all of the products in the database from a result set, not an array list. This method should also print the heading for the list of products. This is a more efficient way to list the products because it doesn't require an array list.
9. To make this work with the ProductDAO interface, which inherits the ProductReader interface, add the definition of the listProducts method to the ProductReader interface. Then, in the main method of the ProductMaintApp class when the "list" action is requested, call the listProducts method from the productDAO object instead of the DisplayAllProducts method. After you compile these classes, test the application. It should work the same as before.

---

## Project 21-1: Display Customer Invoices Report

---

### Console

```
Welcome to the Customer Invoices Report

frankjones@yahoo.com      10504M   11/18/04   $99.00
frankjones@yahoo.com      10501M   10/25/04   $59.50
johnsmith@hotmail.com     10505M   11/18/04   $297.50
johnsmith@hotmail.com     10500M   10/25/04   $495.00
seagreen@levi.com         10502M   10/25/04   $99.00
wendyk@warners.com        10503M   10/25/04   $112.00

Press any key to continue . . .
```

### Operation

- This application connects to a database and displays a list of all customers and their invoices. Each line in this report includes the customer's email address, the invoice number, the invoice date, and the invoice total.

### Specifications

- Register an ODBC Data Source named MurachDB that points to either the Microsoft Access 2000 database file named MurachDB.mdb or the Microsoft Access 97 database file named MurachDB97.mdb. You'll find these files in the `c:\murach\java6\projects\databases` directory. (If you did the exercises for chapter 21, you will have already created this data source.)
- Create a class named `CustomerInvoiceApp` that connects to the database, gets a forward-only, read-only result set that contains the required data, and prints the data in this result set to the console. Each line should include the `EmailAddress` column from the `Customers` table, followed by the `InvoiceNumber`, `InvoiceDate`, and `InvoiceTotal` columns from the `Invoices` table. The rows in the result set should be sorted by the `EmailAddress` column.
- Use a prepared statement to retrieve the data.
- Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named `StringUtils` that has a method that adds the necessary spaces to a string to reach a specified length.
- If the application encounters any exceptions, it should print them to the console.
- When the application finishes, it should close the objects for the result set, the prepared statement, and the database connection.

### Enhancement

- Modify the application so it only displays invoices that have not been processed. To do this, you can add a `where` clause to the SQL statement so it only gets invoices where the `IsProcessed` column is equal to `No`.

---

## Project 21-2: Maintain customer data (JDBC)

---

### Console

```
Welcome to the Customer Maintenance application

COMMAND MENU
list    - List all customers
add     - Add a customer
del     - Delete a customer
help    - Show this menu
exit    - Exit this application

Enter a command: list

CUSTOMER LIST
frankjones@yahoo.com      Frank      Jones
johnsmith@hotmail.com    John       Smith
seagreen@levi.com        Cynthia    Green
wendyk@warners.com       Wendy     Kowolski

Enter a command: add

Enter customer email address: jdbc_test@gmail.com
Enter first name: JDBC
Enter last name: Test

JDBC Test was added to the database.

Enter a command: list

CUSTOMER LIST
frankjones@yahoo.com      Frank      Jones
jdbc_test@gmail.com      JDBC       Test
johnsmith@hotmail.com    John       Smith
seagreen@levi.com        Cynthia    Green
wendyk@warners.com       Wendy     Kowolski

Enter a command: del

Enter customer email to delete: jdbc_test@gmail.com

JDBC Test was deleted from the database.

Enter a command: list

CUSTOMER LIST
frankjones@yahoo.com      Frank      Jones
johnsmith@hotmail.com    John       Smith
seagreen@levi.com        Cynthia    Green
wendyk@warners.com       Wendy     Kowolski

Enter a command: exit

Bye.

Press any key to continue . . .
```

## Operation

- This application begins by displaying a menu with five choices: list, add, del, help, and exit.
- If the user enters “list”, the application displays the customer data that’s stored in a database table.
- If the user enters “add”, the application prompts the user to enter data for a customer and saves that data to the database table.
- If the user enters “del”, the application prompts the user for an email address and deletes the corresponding customer from the database table.
- If the user enters “help”, the application displays the menu again.
- If the user enters “exit”, the application displays a goodbye message and exits.

## Specifications

- Create a class named `Customer` that stores data for the user’s email address, first name, and last name.
- If you haven’t already done so, register an ODBC Data Source named `MurachDB` that points to either the Microsoft Access 2000 database file named `MurachDB.mdb` or the Microsoft Access 97 database file named `MurachDB97.mdb`. You’ll find these files in the `c:\murach\java5\projects\databases` directory. (If you did the exercises for chapter 21, or if you did project 21-1, you will have already created this data source.)
- Create a class named `CustomerDB` that connects to the `MurachDB` database when it’s instantiated. This class should include a public method that returns an array list of `Customer` objects for the customers in the `Customer` table, a public method that returns a `Customer` object for the customer with a specified email address, and public methods that add a record to and delete a record from the `Customer` table.
- Create a `CustomerMaintApp` class that works as shown in the console output. This class should use the `Customer` and `CustomerDB` classes to work with the customer data.
- Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named `StringUtils` that has a method that adds the necessary spaces to a string to reach a specified length.
- Use the `Validator` class or a variation of it to validate the user’s entries. Non-empty strings are required for the email address, first name, and last name.

## Enhancements

- Add an “update” command that lets the user update an existing customer. This command should prompt the user to enter the email address for a customer. Then, it should let the user update the first name and last name for the customer.
- Add a method to the Validator class that uses string parsing techniques to validate the email address. At the least, you can check to make sure that this string contains some text, followed by an @ sign, followed by some more text, followed by a period, followed by some more text. For example, “x@x.x” would be valid while “xxx” or “x@x” would not.
- Use an interface to eliminate any direct calls to the CustomerDB class from the CustomerMaintApp class. To do that, you can use CustomerReader, CustomerWriter, CustomerConstants, and CustomerDAO interfaces as well as a DAOFactory class as described in project 19-3. Then, you can modify the CustomerDB and CustomerMaintApp classes so they use these interfaces and class.

## Chapter 22

# How to work with a Derby database

### Objectives

---

#### Applied

- Use the interactive JDBC tool to connect to a database and execute SQL statements.
- Write the Java code for connecting to and disconnecting from an embedded Derby database.
- Given the specifications for an application that uses an embedded Derby database, develop the application.

#### Knowledge

- Describe the characteristics of a Derby database.
- Explain what is meant by an embedded database.
- Describe the use of SQL scripts.



## Summary

---

- *Apache Derby* is an open-source, Java-based relational database management system (RDBMS) that can be embedded in Java applications or run in a networked client/server system.
- Sun has a distribution of Derby known as *Sun Java DB*. This is the version of Derby that's included in the db directory of Java SE 6. IBM has a distribution of Derby that's known as *IBM Cloudscape*.
- To work with a Derby database, you can use the *ij tool (interactive JDBC tool)* to connect to a database and execute SQL statements.
- SQL statements that have been saved in files are known as *SQL scripts*. You can run these scripts from the *ij* prompt or the command prompt.
- If you connect to a Derby database with a username and create a database object, that database will be stored in a *schema* that corresponds to the username. Then, to access that object, you need to connect to the database with the same username or to qualify any reference to that object with the username.
- An *embedded Derby database* runs in the same process in which the application that uses the database runs. An embedded database is normally stored in the same directory as the Java class that uses it.
- You can also run a Derby database management system on a network server so the database can be accessed by two or more clients. In this case, the URL for connecting to a database must include the server name and port.

## Terms

---

Apache Derby  
Sun Java DB  
IBM Cloudscape  
Derby database  
interactive JDBC tool (ij tool)  
SQL script  
embedded Derby database

## **Exercise 22-1    Create and access a Derby database**

---

In this exercise, you'll use JDBC code to create the MurachDB database as a Derby database. Then, you'll use the ij tool to work with that database.

### **Create the Derby database with a Java application**

1. Add the derby.jar and derbytools.jar files to the class path as described in figure 22-2.
2. Open the MurachDB class that's in the ch22\DBTester directory. Review the code for this class. It contains all of the code necessary to create an embedded Derby database named MurachDB, create the Products table, and insert two rows into this table. Note that you don't need to call the setDbDirectory method if you're going to store the database in the same directory as the application. Also note that this code specifies empty strings for the username and password for the database. As a result, this code uses the same database schema that's used when you use the ij tool to connect to the database.
3. Open the DBTesterApp class that's in the ch22\DBTester directory. Review the code and then run the application. It should create the Derby version of the MurachDB database and print all of the records in the Products table to the console. Then, it should print several blank product lines, print a record to be inserted into the database, print the all the records again, print a record to be deleted from the database, and print all the records again.

### **Complete the methods in the MurachDB class**

4. Write the code for the printFirstProduct, printLastProduct, printProductByCode, insertProduct, and deleteProduct methods in the MurachDB class. If you completed exercise 21-1, you can copy the code from that solution. Otherwise, you can refer to chapter 21 and exercise 21-1 for details on how to implement these methods. This shows that the JDBC code works the same for an Access database or a Derby database. The main difference is how you create and connect to the database.

### **Use the ij tool to work with the Derby database**

5. Use the ij tool to connect to the MurachDB database as shown in figure 22-3.
6. Run a SELECT statement that selects all records from the Products table as shown in figure 22-4.
7. Run an INSERT statement that adds a row to the Products table as shown in figure 22-4, and run the SELECT statement of step 6 again.
8. Disconnect from the database and exit the ij tool.

## **Exercise 22-2     Use a Derby database with the Product Maintenance application**

---

This exercise has you modify the Product Maintenance application of exercise 21-2 so it uses a Derby database instead of an Access database. This is the same application that was originally presented in chapter 8.

### **Create an embedded Derby database**

1. Open the DBTesterApp class that's in ch22\ProductMaintenance directory. This is an abridged version of the DBTesterApp class that you used in exercise 22-1. This abridged version uses the methods in the MurachDB class just to create a Derby database in the ProductMaintenance directory. Run this application to create the embedded MurachDB database, and then close this class.

### **Modify the Product Maintenance application so it uses that database**

2. Open the ProductDB class that's in ch22\ProductMaintenance directory and review the code for this class. Note that the connect method connects to the Access database named MurachDB, because this is the same class that was used for the exercises in chapter 21. Now, modify this code so it connects to the MurachDB database that you created in step 1. Use empty strings for the username and password so you can use the ij tool to connect to this database later on.
3. Open the ProductMaintApp class and run it. Everything should work the same as it did with the Access version of this application. Note, however, that this application doesn't disconnect from the database when the application ends.
4. Add a disconnect method to the ProductDB class that shuts down the Derby database engine, and modify the ProductMaintApp class so it calls this method when the application ends. Then, test this enhancement.

---

## Project 22-1: Display Customer Invoices Report (Derby)

---

### Description

- If you've already done project 21-1, copy and modify it so it uses a Derby database. If you haven't done project 21-1, do it with a Derby database.

### Specifications

- Create a class named `MurachDB` that has the methods for creating an embedded Derby database. You can find the SQL statements for creating this database in the `CreateMurachDB.sql` file that's stored in the `c:\murach\java6\Projects\Derby` directory.
- Modify the `CustomerInvoiceApp` class described in project 21-1 so it uses the `MurachDB` class to create the embedded database if it doesn't already exist. Include statements that print messages to the console that indicate (1) when Derby is starting and shutting down, (2) when a table is being created, and (3) when rows are being inserted into the database.
- After the database has been created, the application should do the tasks described in project 21-1.

---

## Project 22-2: Maintain customer data (Derby)

---

### Description

- If you've already done project 21-2, copy and modify it so it uses a Derby database. If you haven't done project 22-1, do it with a Derby database.

### Specifications

- Copy the MurachDB class from your solution for project 22-1 if you've done that project. Otherwise, create a class named MurachDB that has the methods for creating an embedded Derby database. You can find the SQL statements for creating this database in the CreateMurachDB.sql file that's stored in the `c:\murach\java6\Projects\Derby` directory.
- Modify the CustomerMaintApp class described in project 21-2 so it uses the MurachDB class to create the embedded database if the database doesn't already exist.
- After the database has been created, the application should do the tasks described in project 21-2.