UMBC
Training Centers

ADVANCED JAVA

Course # TCPRG9902
Rev. 3/14/2016

UMBC
Training Centers
6996 Columbia Gateway Drive
Suite 100
Columbia, MD 21046
Tel: 443-692-6600
http://www.umbctraining.com

ADVANCED JAVA

Course # TCPRG9902
Rev. 3/14/2016

This Page Intentionally Left Blank

Course Objectives

- At the conclusion of this course, students will be able to:
  - Document and package a Java application.
  - Use many of the new enhancements added to the Java API.
  - Use assertions to write robust Java code.
  - Use regular expressions for efficient pattern matching.
  - Choose appropriate data structures from the Java Collection API.
  - Sort and search arrays and lists using a variety of techniques.
  - Capture configuration and debugging information using the Java Logging APIs.
  - Use Generics to create type safe collections.
  - Serialize Java objects.
  - Reading and Writing XML
  - Use the Java Security API
  - Write TCP/IP Client Server applications using sockets.
  - Write multi-threaded Java applications.
  - Execute methods on a remote object using RMI.
  - Perform database queries and updates using JDBC.

This Page Intentionally Left Blank

Table of Contents

CHAPTER 1: REVIEW OF JAVA FUNDAMENTALS
The Java Environment 1-2
Data Types 1-6
The `String` Class 1-7
The `StringBuffer` Class 1-9
Arrays 1-10
Passing Data Types to a Method 1-12
Constructors and Initialization 1-13
Inheritance 1-16
Abstract Classes 1-18
Interfaces 1-21
Static Data, Methods, and Blocks 1-25
Wrapper Classes 1-27
I/O 1-28

CHAPTER 2: PACKAGING AND DISTRIBUTING A JAVA APPLICATION
Packages 2-2
Managing Source and Class Files 2-4
The `javadoc` Utility 2-10
Documenting Classes and Interfaces 2-12
Documenting Fields 2-13
Documenting Constructors and Methods 2-14
Running the `javadoc` Utility 2-16
`jar` Files 2-17
The Manifest File 2-19
Bundling and Using Jar-Packaged Resources 2-20

CHAPTER 3: MISCELLANEOUS ENHANCEMENTS
Enhanced `for` Loop 3-2
Autoboxing and Auto-Unboxing 3-4
Static Imports 3-6
`varArgs` 3-8
Typesafe Enums 3-12
Formatted Strings 3-16
Format Specifier Syntax 3-17
Format Specifier Conversions 3-18
Format Specifier Flags 3-22
Formatted Integers Example 3-23
Formatted Floating Points Example 3-24
Formatted Strings Example 3-25
Formatted Dates Example 3-26
Complex Formatted Example 3-27

CHAPTER 4: ASSERTIONS
Introduction 4-2

Assertion Syntax 4-3
Compiling with Assertions 4-5
Enabling and Disabling Assertions 4-6
Assertion Usage 4-9

CHAPTER 5: REGULAR EXPRESSIONS
Regular Expressions 5-2
String Literals 5-4
Character Classes 5-5
Quantifiers 5-9
Capturing Groups and Backreferences 5-11
Boundary Matchers 5-12
Pattern and Matcher 5-13

CHAPTER 6: THE JAVA COLLECTION CLASSES
Introduction 6-2
The Arrays Class 6-3
Searching and Sorting Arrays of Primitives 6-4
Sorting Arrays of Objects 6-5
The `Comparable` and `Comparator` Interfaces 6-6
Sorting - Using `Comparable` 6-7
Sorting - Using `Comparator` 6-10
Collections 6-11
Lists and Sets 6-12
Iterators 6-13
Lists and Iterators Example 6-14
Maps 6-17
Maps and Iterators Example 6-18
The Collections Class 6-20
Rules of Thumb 6-23

CHAPTER 7: GENERICS
Introduction 7-2
Defining Simple Generics 7-4
Generics and Subtyping 7-5
Wildcards 7-6
Bounded Wildcards 7-8
Generic Methods 7-9

CHAPTER 8: ADVANCED I/O
Introduction 8-2
Basic File I/O Example 8-4
Buffered I/O 8-5
The `Console` Class 8-8
Object Serialization 8-10
Serialization Issues 8-16
Compressed Files 8-18
Zip File Example 8-19

Writing Your Own I/O Classes 8-21
Property Files 8-23
The `Preferences` Class 8-26

CHAPTER 9: JAVA SECURITY
Security Overview 9-2
Java Language Security 9-4
Access Control 9-6
Policies and Permissions 9-9
Property Permissions 9-10
File Permissions 9-11
Socket Permissions 9-13
Policy Files 9-14
Encrypting Important Data 9-17
Ciphers 9-20
Summary 9-23

CHAPTER 10: LOGGING API
Introduction 10-2
Loggers 10-3
Logger Levels 10-5
Logger Handlers 10-6
Specifying Handlers and Formatters 10-9
Configuring Handlers 10-10
LogManager 10-13

CHAPTER 11: NETWORKING
Networking Fundamentals 11-2
The Client/Server Model 11-4
`InetAddress` 11-6
URLs 11-8
Sockets 11-11
A Time-of-Day Client 11-13
Writing Servers 11-14
Client/Server Example 11-15

CHAPTER 12: THREADS AND CONCURRENCY
Review of Fundamentals 12-2
Creating Threads by Extending `Thread` 12-3
Creating Threads by Implementing `Runnable` 12-4
Advantages of Using Threads 12-5
Daemon Threads 12-8
Thread States 12-10
Thread Problems 12-14
Synchronization 12-16
Performance Issues 12-19

CHAPTER 13: REMOTE METHOD INVOCATION (RMI)
Introduction 13-2

RMI Architecture 13-3
The Remote Interface 13-4
The Remote Object 13-5
Writing the Server 13-6
The RMI Compiler 13-8
Writing the Client 13-9
Remote Method Arguments and Return Values 13-10
Dynamic Loading of Stub Classes 13-11
Remote RMI Client Example 13-12
Running the Remote RMI Client Example 13-20

CHAPTER 14: JAVA DATABASE CONNECTIVITY (JDBC)

Introduction 14-2
Relational Databases 14-4
Structured Query Language 14-5
A Sample Program 14-6
Transactions 14-9
Meta Data 14-15

Chapter 1:
Review of Java Fundamentals

1) The Java Environment 1-2

2) Data Types 1-6

3) The `String` Class 1-7

4) The `StringBuffer` Class 1-9

5) Arrays 1-10

6) Passing Data Types to a Method 1-12

7) Constructors and Initialization 1-13

8) Inheritance 1-16

9) Abstract Classes 1-18

10) Interfaces 1-21

11) Static Data, Methods, and Blocks 1-25

12) Wrapper Classes 1-27

13) I/O 1-28

The Java Environment

- In any Java project, it is often necessary to set up the environment in which you will be working.
  - This includes, but is not limited to where the:
    - Java SDK is installed;
    - source files are located;
    - compiled `.class` files reside; and
    - third party class files (if necessary) are located.
- Although there are many ways to set up your environment, the approach taken for this course specifies the necessary information in a script named `setenv.cmd`.
  - Having all of the settings in a single file can make changing the environment easier.
- One of the most common environment variables is `PATH`.
  - Including the `bin` directory of the Java SDK in the `PATH` permits the user to run the `java` and/or `javac` executables on the command line, without having to supply the full path to the executables every time.
- Two other common environment variables for Java are `JAVA_HOME` and `CLASSPATH`.
  - The `JAVA_HOME` variable references the directory in which Java was installed.
  - The `CLASSPATH` variable can be used to permit the `java` and `javac` executables to easily locate the `.class` files.

The Java Environment

- In addition to the common variables defined above, it is convenient to define custom variables that can be used as shortcuts.
  - In addition to the standard environment variables, this course will define the following custom variables.

Variable	Value
LABDIR	The location of the lab files for the course
SOURCE	The location of the source .java files
CLASSES	The location of the compiled .class files
DOCS	The location of the javadocs specific to the course

- The instructor will inform you which directory contains the lab files for this course.
  - You may wish to write its location here as reference.

    LABDIR = _____

- The only values that might need changing in the setenv script are the values for JAVA_HOME and LABDIR.

setenv.cmd

```
 1.  @ECHO OFF
 2.  REM SET Lab Specific Environment Variables
 3.  set LABDIR=C:\AdvancedJavaProject
 4.  set SOURCE=%LABDIR%\src
 5.  set CLASSES=%LABDIR%\bin
 6.  set DOCS=%LABDIR%\docs
 7.
 8.  REM SET PATH and CLASSPATH Environment Variables
 9.  set JAVA_HOME=C:\javasoft\jdk1.5.0_10
10.  set PATH=%JAVA_HOME%\bin;%PATH%
11.  set CLASSPATH=%LABDIR%
```

The Java Environment

- The `setenv.cmd` script is located in the `LABDIR` directory.

- The lab files for this course are arranged so that the `.class` files are created in a separate directory from the `.java` files.
  - The full benefit of working in such an environment will be discussed in detail in the next chapter.
  - The example that follows demonstrates using the `-d` option of the compiler to redirect the resultant `.class` files to a specific directory.
    - The specified location must be listed on the `CLASSPATH`.

HelloWorld.java

```
1.  package examples.review;
2.  public class HelloWorld {
3.      public static void main(String[] args) {
4.          System.out.println("Hello World");
5.      }
6.  }
```

- The steps needed to compile and execute the above program are shown on the following page.

The Java Environment

- The steps needed to compile and execute the above program are listed below.
  - Open a new command prompt.
  - Change the directory to the lab files directory, and enter `setenv` to set the environment.
  - Change into the subdirectory `src\examples\review`.
    - This directory has the source file `HelloWorld.java`.
  - Compile the Java code to the `bin` directory of the lab files by utilizing the `-d` option of the compiler as shown below.

    ```
    javac -d %CLASSES% HelloWorld.java
    ```

    - Note that the `-d` option used above will create the necessary `examples\review` directory structure within the `bin` directory and place the `.class` file in the resultant directory.
  - Execute the Java program by supplying its fully qualified class name as shown below.

    ```
    java examples.packages.HelloWorld
    ```
- The above technique for compiling and executing the Java code will continue to be used throughout the course.

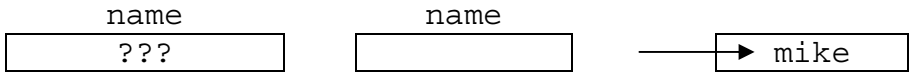
Data Types

- A variable is an instance of a data type that is referred to by an **identifier**.
  - An **identifier** can be any length, beginning with a letter and consisting of letters, digits, the `$`, and the underscore "_" character.
- Java has two varieties of data types - **primitive** and **reference** types.
  - A data type is a specification for a named storage area in memory that can store a set of legal values and have a set of operations performed on values of this type.
  - The **primitive** data types are listed below.
    - `byte`
    - `short`
    - `int`
    - `long`
    - `float`
    - `double`
    - `char`
    - `boolean`
  - A **reference** type is used for all arrays and user-defined data types (such as classes and interfaces).
    - The `String` class is an example of a reference type.
    - A reference can have the value `null` or can refer to an instantiated object.
    - Java objects are instantiated with the `new` keyword.
- The Java SE library ships with a multitude of pre-built classes, each representing a different data type.

The `String` Class

- Variables store references to `String` objects (like all objects in Java) rather than the actual object.

```
String name;        name = new String("mike");
```



- A `String` object can be obtained by using the `new` operator to explicitly create the object, or through using a literal `String` as shown below.

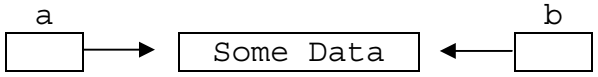
```
String x = new String("Michael Saltzman");
String y = "Hello World";
```

  - Keep in mind that the `String` class is the only one that does not require the use of the `new` keyword to create an object.
    - When creating a `String` with the `new` keyword, a new object is always created.

```
String x = new String("Some Data");
String y = new String("Some Data");
```

    - Creating a literal String object does not always result in the creation of a new object.

```
String a = "Some Data";
String b = "Some Data";
```



- A `String` object, once instantiated, cannot be changed. It is referred to as **immutable**.

The `String` Class

StringTest.java

```
 1. package examples.review;
 2. public class StringTest {
 3.     public static void main(String args[]) {
 4.
 5.         String s = new String("123456789ABCABC");
 6.         print("Data is: " + s);
 7.         print("Length is:" + s.length());
 8.
 9.         int x = s.indexOf('A');
10.         print("The letter A was found at position "
11.                 + x + " and position " +
12.                 s.indexOf('A', x + 1));
13.
14.         print("Last occurrence of the letter C "
15.                 + "was found at position " +
16.                 s.lastIndexOf('C'));
17.
18.         print("substring(0, 9): " +
19.                 s.substring(0, 9));
20.
21.         print("substring(9): " + s.substring(9));
22.
23.         print("toLowerCase(): " + s.toLowerCase());
24.     }
25.     public static void print(String s){
26.          System.out.println(s);
27.     }
28. }
```

- Running the example above creates the following output.

```
Data is: 123456789ABCABC
Length is:15
The letter A was found at position 9 and position 12
Last occurrence of the letter C was found at position 14
substring(0, 9): 123456789
substring(9): ABCABC
toLowerCase(): 123456789abcabc
```

The `StringBuffer` Class

- The contents of a `StringBuffer` can be altered through various method calls as shown below.

StringBufferTest.java

```
 1. package examples.review;
 2. public class StringBufferTest{
 3.     public static void main(String args[]){
 4.         StringBuffer sb = new StringBuffer();
 5.         info(sb);
 6.         info(sb.append(123456789));
 7.         info(sb.insert(0, "abcdefghi"));
 8.         info(sb.replace(2, 5, "Hello"));
 9.         sb.setLength(0);
10.         info(sb);
11.     }
12.     public static void info(StringBuffer sb){
13.         System.out.print("len:" + sb.length());
14.         System.out.print(" cap:" + sb.capacity());
15.         System.out.print(" data: " + sb);
16.         System.out.println();
17.     }
18. }
```

- A `StringBuffer` is often used to avoid the overhead associated with `String` concatenation.

- Another way to avoid `String` concatenation without the use of a `StringBuffer` is by rewriting each of the `System.out.print()` methods in the above code as two statements. This is shown below.

```
System.out.print("len:");
System.out.print(sb.length());
```

Arrays

- An array is an ordered collection of data items, all of whose types are the same.
  - The reference to an array can be declared in either of the following ways.
    - *datatype variableName*`[];` e.g. `int x [];`
    - *datatype* `[]` *variableName*`;` e.g. `String [] s;`
  - The `new` operator is used to instantiate an array (like any other object).

```
x = new int[4];
s = new String[3];
```

  - The declaration and initialization could be combined as:

```
int x [] = new int[4];
String [] s = new String[3];
```

  - An array can also be declared and initialized with shortcut syntax as shown below.

```
int x [] = {0, 1, 2, 3};
String [] s = {"Michael", "Susan", "Alan"};
```

- Since an array is an object, it has certain properties.
  - The `length` property is provided for all arrays.
    - `length` is a property (not a method), so it is not followed by a set of parenthesis.

```
int len = x.length; //length of the array
```

- Arrays are typically processed with loops, as demonstrated in the example on the following page.

Arrays

- The example below demonstrates various techniques for working with arrays.

ArrayTests.java

```
 1. package examples.review;
 2. import java.util.Random;
 3.
 4. public class ArrayTests {
 5.     public static void main(String args[]) {
 6.         // Loop through command line arguments
 7.         for (int i = 0; i < args.length; i++)
 8.             System.out.println(args[i]);
 9.
10.         // Fill an array with random numbers
11.         // from 0 to 99
12.         int x [] = new int[10];
13.         Random r = new Random();
14.         for(int i = 0; i < x.length; i++){
15.             x[i] = r.nextInt(100);
16.         }
17.
18.         // Print contents of array and calculate
19.         // total of numbers in the array
20.         int total = 0;
21.         for(int i = 0; i < x.length; i++){
22.             System.out.print(x[i] + " ");
23.             total += x[i];
24.         }
25.         System.out.println("Total is " + total);
26.
27.         // Use shorthand notation to create an array
28.         String [] days = {"Sun", "Mon", "Tue", "Wed",
29.                 "Thu", "Fri", "Sat"};
30.         for(int i = 0; i < days.length; i++)
31.             System.out.print(days[i] + " ");
32.     }
33. }
```

Passing Data Types to a Method

- Whenever a data type is passed to a method, it is always passed by value.
  - For primitive data types, a copy of the actual data type is passed.
  - For reference data types, a copy of the reference is passed, not the actual object.
- A primitive can be passed by reference by wrapping it inside of an object (such as an array of one element), as demonstrated below.

PassingPrimitives.java

```
 1. package examples.review;
 2. public class PassingPrimitives {
 3.     public static void main(String[] args) {
 4.         // change method cannot modify primitive data
 5.         // being passed as argument to method
 6.         int x = 10;
 7.         change(x);
 8.         System.out.println(x);
 9.
10.         // change method can modify object whose
11.         // reference is passed as argument to method
12.         int data[] = {x};
13.         change(data);
14.         System.out.println(data[0]);
15.     }
16.     public static void change(int p) { p = 20; }
17.
18.     public static void change(int p[]) { p[0] = 20; }
19. }
```

Constructors and Initialization

- Constructors cannot have a return type, not even `void`.
  - Note what happens when the following program is run.

Person.java

```
 1.  package examples.review;
 2.
 3.  public class Person {
 4.      protected String firstName;
 5.      protected String lastName;
 6.
 7.      public void Person(){
 8.          firstName = "Jane";
 9.          lastName = "Doe";
10.      }
11.
12.      public static void main(String[] args) {
13.          Person p = new Person();
14.          System.out.println(p.firstName);
15.          System.out.println(p.lastName);
16.      }
17.  }
```

- Instance data is automatically initialized in Java. Therefore, the task of a constructor is not necessarily initialization.
  - **Numerical** data is initialized to `zero`.
  - **Boolean** data is initialized to `false`.
  - **Object** reference data is initialized to `null`.
- Instance data can also be explicitly initialized when it is declared.

Constructors and Initialization

- The class below contains several fields (instance data) demonstrating the implicit and explicit initialization that will occur during its construction.

SimpleClass.java

```
 1. package examples.review;
 2.
 3. public class SimpleClass {
 4.     private boolean a, b = true;
 5.     private int x, y = 99;
 6.     private String s, t = "Hello";
 7.
 8.     public String toString(){
 9.         StringBuffer sb = new StringBuffer();
10.         sb.append(a).append(':').append(b);
11.         sb.append('\t');
12.         sb.append(x).append(':').append(y);
13.         sb.append('\t');
14.         sb.append(s).append(':').append(t);
15.         return sb.toString();
16.     }
17.
18.     public static void main(String[] args) {
19.         SimpleClass sc = new SimpleClass();
20.         System.out.println(sc);
21.     }
22. }
```

- The output of the above program is shown below.

```
false:true    0:99    null:Hello
```

- When a class has a set of overloaded constructors, it is preferred that the parameters passed to one of the constructors be handled by another constructor, as demonstrated in the example on the next page.

Constructors and Initialization

Circle.java

```
 1. package examples.review;
 2. public class Circle {
 3.     int xc, yc, radius;
 4.
 5.     public Circle(int xc, int yc, int rad) {
 6.         this.xc = xc;
 7.         this.yc = yc;
 8.         this.radius = rad;
 9.     }
10.     public Circle(int x, int y) {
11.         this(x, y, 1);
12.     }
13.     public Circle(int rad) {
14.         this(0, 0, rad);
15.     }
16.     public Circle() {
17.         this(0, 0, 1);
18.     }
19. }
```

- The use of the functional form "`this()`" is only permitted inside of a constructor and must be the first statement in the constructor.

- The use of keyword `this` as a reference to the "host object" is optional, unless the parameter being passed to a method of the class, or one of its constructors is named the same as one of the instance variables of the class.
  - This can be seen in the example above for the constructor that takes three `int` parameters.
    - `this.xc = xc;` - use is necessary
    - `this.yc = yc;` - use is necessary
    - `this.radius = rad;` - use is optional

Inheritance

- Often, a class is needed, which is a specialization of an existing class.
  - By using the keyword `extends`, a specialized class can be built so that the methods and data of an existing class can be reused without being re-coded.
  - The new class can add or override methods to implement its special behavior. Data can also be added to the new class.
- The building of classes from existing classes in this way is called **inheritance**.
  - The relationship between the existing class (superclass) and the newly created class (subclass) is called an **is-a** relationship.
  - The process can be repeated, giving rise to a hierarchy of classes originating from one class.
  - All functionality from the superclass is reusable in the newly extended subclass.
  - The `final` keyword can be used in a class definition to prevent a class from being extended.
- Similar to the keyword `this`, two forms of the keyword `super` exist.
  - `super()` can be used in a constructor to pass parameters to a constructor in the superclass.
  - `super` acts a reference to the instance methods and fields of the superclass.

Inheritance

- A subclass is responsible for calling a constructor from its superclass using `super(`*`params`*`)`, where *`params`* represents the parameters required in the superclass constructor.
  - If the use of `super()` is omitted, the compiler will automatically insert a call to the no-argument constructor in the superclass.
    - The subclass will not compile if the superclass does not have a no-argument constructor.
- The example below attempts to define two classes that form an is-a relationship.
  - As written, `SuperClass.java` will compile, whereas `SubClass.java` will not compile.

SuperClass.java

```
1.  package examples.review;
2.  public class SuperClass {
3.      int x;
4.      public SuperClass(int x){
5.          this.x = x;
6.          System.out.println("In Super Constructor");
7.      }
8.  }
```

SubClass.java

```
1.  package examples.review;
2.  public class SubClass extends SuperClass{
3.      public SubClass(){
4.          System.out.println("SubClass Constructor");
5.      }
6.  }
```

Abstract Classes

- An abstract class represents an abstract concept. Therefore, it cannot be instantiated but must be subclassed.
  - In addition to the methods and data inside of a class, an abstract class may consist of one or more methods that are abstract and therefore, describe a programming interface.
  - The implementation for these abstract methods is left to classes that extend it.
- The example below defines an abstract class named `AbstractDate`.
  - The class has a method named `isLeapYear` that is inherited by any subclass of `AbstractDate`.
  - The `abstract` methods named `getDay, getMonth,` and `getYear` must be implemented with any subclass of `AbstractDate`.
  - The class leaves it up to the subclass to define how the actual data is stored and from where the data comes.

AbstractDate.java

```
 1. package examples.review;
 2. public abstract class AbstractDate{
 3.
 4.     public boolean isLeapYear() {
 5.         int y = getYear();
 6.         boolean success = false;
 7.         if(y % 400 == 0 ||
 8.             ((y % 4 == 0) && (y % 100 != 0)))
 9.             success =  true;
10.         return success;
11.     }
12.     public abstract int getDay();
13.     public abstract int getMonth();
14.     public abstract int getYear();
15. }
```

Abstract Classes

- Below are two substantially different implementations of the abstract class named `AbstractDate`.

SimpleDate.java

```
 1. package examples.review;
 2.
 3. public class SimpleDate extends AbstractDate {
 4.     int data [] = new int[3];
 5.     public SimpleDate(int month, int day, int year){
 6.         data[0] = day;
 7.         data[1] = month;
 8.         data[2] = year;
 9.     }
10.     public int getDay() {  return data[0]; }
11.
12.     public int getMonth() { return data[1]; }
13.
14.     public int getYear() { return data[2]; }
15. }
```

Today.java

```
 1. package examples.review;
 2. import java.util.Calendar;
 3.
 4. public class Today extends AbstractDate {
 5.     public int getDay() {
 6.        Calendar c = Calendar.getInstance();
 7.        return c.get(Calendar.DAY_OF_MONTH);
 8.     }
 9.
10.     public int getMonth() {
11.         Calendar c = Calendar.getInstance();
12.         return c.get(Calendar.MONTH) + 1;
13.     }
14.
15.     public int getYear() {
16.         Calendar c = Calendar.getInstance();
17.         return c.get(Calendar.YEAR);
18.     }
19. }
```

Abstract Classes

- The example below shows how both of the implementations of the abstract class can be referenced by a single variable of type `AbstractDate`.

DateTesting.java

```
 1. package examples.review;
 2.
 3. public class DateTesting {
 4.
 5.     public static void main(String[] args) {
 6.         showDateInfo(new SimpleDate(1, 31, 2008));
 7.         showDateInfo(new Today());
 8.     }
 9.     public static void showDateInfo(AbstractDate d){
10.         System.out.print(d.getMonth());
11.         System.out.print("/");
12.         System.out.print(d.getDay());
13.         System.out.print("/");
14.         System.out.println(d.getYear());
15.     }
16. }
```

Interfaces

- An **interface** is a collection of abstract methods and possibly some static constant values.
  - Interfaces are similar to abstract classes, except that they can be implemented by a set of classes not related in the same inheritance hierarchy.
    - Whereas abstract classes are typically used to represent objects whose implementations share a lot of commonality, interfaces are typically used to allow objects having nothing in common to interact with an application in a standardized way.
  - Although a class cannot extend more than one class, it can implement more than one interface.
  - It is also possible for a class to extend one class and implement one or more interfaces.

- The following example is an application that relies on the interface shown below to notify listeners of a change in temperature.

TemperatureListener.java

```
1.  package examples.review.threads;
2.
3.  public interface TemperatureListener {
4.      public void temperatureChanged(int temperature);
5.  }
```

  - Any object that implements the above interface will be informed of a change in temperature through the `temperatureChanged` method.
    - When the temperature changes, what action to take is left up to the implementing class, as demonstrated in the code that follows.

Interfaces

- The class below is a `Thread` that relies upon the interface defined on the previous page.

WeatherBroadcast.java

```
 1. package examples.review.threads;
 2. import java.util.*;
 3.
 4. public class WeatherBroadcast extends Thread {
 5.     private Vector listeners = new Vector();
 6.     private int tmp = 70;
 7.
 8.     public void addListener(TemperatureListener t){
 9.         listeners.add(t);
10.     }
11.
12.     public void run() {
13.         while (true){
14.             int direction = new Random().nextInt(2);
15.             int amount = new Random().nextInt(15);
16.             for (int i = 0; i < amount; i++) {
17.                 if(direction == 0)
18.                     tmp++;
19.                 else
20.                     tmp--;
21.                 Enumeration e = listeners.elements();
22.                 TemperatureListener listener;
23.                 while(e.hasMoreElements()){
24.                    Object obj = e.nextElement();
25.                    listener =
26.                        (TemperatureListener) obj;
27.                    listener.temperatureChanged(tmp);
28.                 }
29.                 try {
30.                     Thread.sleep(1000);
31.                 } catch (InterruptedException ie) {
32.                     //quietly ignore
33.                 }
34.             }
35.         }
36.     }
37. }
```

Interfaces

- The two examples below implement the `TemperatureListener` interface in different ways.
  - The first implementation simply prints out the temperature every time it changes.

SampleListener1.java

```
 1. package examples.review.threads;
 2.
 3. public class SampleListener1 implements
 4.         TemperatureListener {
 5.     public void temperatureChanged(int temperature) {
 6.         System.out.print("Current Temperature is ");
 7.         System.out.println(temperature + "\u00B0");
 8.     }
 9. }
```

  - The second implementation prints a message every time there is a five degree change in temperature.

SampleListener2.java

```
 1. package examples.review.threads;
 2. public class SampleListener2 implements
 3.         TemperatureListener {
 4.     int startTemp;
 5.     boolean isValid = false;
 6.     String msg = "There has been a 5\u00B0 change.";
 7.
 8.     public void temperatureChanged(int temperature) {
 9.         if(!isValid){
10.             startTemp = temperature;
11.             isValid = true;
12.         }
13.         int diff = Math.abs(temperature - startTemp);
14.         if( diff >= 5){
15.             System.out.println(msg);
16.             startTemp = temperature;
17.         }
18.     }
19. }
```

Interfaces

- The code below combines the data types from the previous pages into an application demonstrating the use of the classes that implement the interface.

WeatherTest.java

```
 1. package examples.review.threads;
 2.
 3. public class WeatherTest {
 4.
 5.     public static void main(String notused[]) {
 6.         WeatherBroadcast ws = new WeatherBroadcast();
 7.         ws.setDaemon(true);
 8.         ws.addListener(new SampleListener1());
 9.         ws.addListener(new SampleListener2());
10.         ws.start();
11.         try {
12.             Thread.sleep(30000);
13.         } catch (InterruptedException e) {
14.             e.printStackTrace();
15.         }
16.     }
17. }
```

  - The `setDaemon(true)` is called on the WeatherBroadcast thread so that it does not prevent the application from terminating at the end of the `main` method.

Static Data, Methods, and Blocks

- In Java, static data is used to represent class data, as opposed to instance data.
  - Each object will have its own copy of instance data, but there will be only one copy of class data, shared by all the objects.
- Static data is often used for generated numbers, which can later be used to uniquely identify an object.
  - Static methods are often used to access static data.

TestStudent.java

```
 1. package examples.review;
 2. public class TestStudent {
 3.     public static void main(String argv[]) {
 4.         System.out.println(Student.getNextID());
 5.         Student s[] = { new Student(), new Student(),
 6.                 new Student(), new Student() };
 7.         for (int i = 0; i < s.length; i++)
 8.             System.out.println(s[i].getID());
 9.         System.out.println(Student.getNextID());
10.     }
11. }
12. class Student {
13.     private static int idGenerator;
14.     private int studentID;
15.
16.     public Student() {
17.         studentID = idGenerator++;
18.     }
19.
20.     public int getID() {
21.         return studentID;
22.     }
23.
24.     public static int getNextID() {
25.         return idGenerator;
26.     }
27. }
```

Static Data, Methods, and Blocks

- Although you may be familiar with the previous uses of `static`, you may not be familiar with a **static block**.
  - A block of code that is prefixed with the word `static` is executed on behalf of the class, when the class is loaded by the Java Virtual Machine.
  - A static block is useful when objects need some common service that must be present when the objects are instantiated.
    - Since a class is only loaded once, a static block of code will only be executed once.
- Below is an example of the syntax for a `static` block.

```
 1. package examples.review;
 2.
 3. public class StaticBlock {
 4.     static {
 5.         System.out.println("StaticBlock loaded");
 6.     }
 7.     public static void main(String args[]) {
 8.         new One();
 9.         new One();
10.     }
11. }
12.
13. class One {
14.     static {
15.         System.out.println("One has been loaded");
16.     }
17.     public One() {
18.         System.out.println("One's constructor");
19.     }
20.     static {
21.         System.out.println("Additional Output");
22.     }
23. }
```

Wrapper Classes

- There is a set of classes, referred to as the wrapper classes, which provide:
  - object versions of the primitive data types; and
  - static methods to convert a primitive into a `String` and a `String` into a primitive.
    - The class names are based on the primitive data type names and are listed below.

      ```
      Byte     Double    Float     Integer
      Long     Short     Boolean   Character
      ```

- The example below demonstrates the various uses of the `Integer` class.

WrapperClasses.java

```
 1. package examples.review;
 2. import java.util.Enumeration;
 3. public class WrapperClasses {
 4.     public static void main(String[] args){
 5.         int sum = 0;
 6.         java.util.Vector v = new java.util.Vector();
 7.         v.add(new Integer(10));
 8.         v.add(new Integer(20));
 9.         Enumeration e = v.elements();
10.         while(e.hasMoreElements()){
11.             Integer temp = (Integer) e.nextElement();
12.             sum += temp.intValue();
13.         }
14.         System.out.println("Total is " + sum);
15.
16.         sum = 0;
17.         for(int i = 0; i < args.length; i++){
18.             sum += Integer.parseInt(args[i]);
19.         }
20.         System.out.println("Total is " + sum);
21.     }
22. }
```

I/O

- The example that follows reads from a file and converts each line of text found in the file to a `Calendar` object.
  - Each `Calendar` object is then formatted and written to a different file.
  - The application also relies on a set of utility functions to perform the closing of the input and output streams.

IOExample.java

```
 1.  package examples.review;
 2.  import java.io.*;
 3.  import java.text.SimpleDateFormat;
 4.  import java.util.*;
 5.
 6.  public class IOExample {
 7.
 8.      public static void main(String[] args) {
 9.          BufferedReader br = null;
10.          PrintWriter pw = null;
11.          String theLine;
12.          Date d;
13.          String filePath = "src/examples/review/";
14.          String fileName = filePath + "dates.txt";
15.          String output = filePath + "output.txt";
16.          try {
17.              FileReader fr =
18.                  new FileReader(fileName);
19.              br = new BufferedReader(fr);
20.              pw = new PrintWriter(output);
21.              while((theLine = br.readLine()) != null){
22.                  d = parseAsDate(theLine);
23.                  printDate(pw, d);
24.              }
25.          } catch (IOException e) {
26.              e.printStackTrace();
27.          } finally {
28.              IOCleanupUtils.cleanup(br);
29.              IOCleanupUtils.cleanup(pw);
30.          }
31.      }
```

I/O

IOExample.java - *continued*

```
32.
33.     final static SimpleDateFormat sdf1 =
34.         new SimpleDateFormat("MM/dd/yyyy");
35.     final static SimpleDateFormat sdf2 =
36.         new SimpleDateFormat("EEEE");
37.
38.     public static Date parseAsDate(String text){
39.         String s [] = text.split("/");
40.         int m = Integer.parseInt(s[0]) - 1;
41.         int d = Integer.parseInt(s[1]);
42.         int y = Integer.parseInt(s[2]);
43.         GregorianCalendar gc =
44.             new GregorianCalendar(y, m, d);
45.         return gc.getTime();
46.     }
47.
48.     public static void printDate(PrintWriter pw,
49.                                  Date d){
50.         pw.print(sdf1.format(d));
51.         pw.print(" is a ");
52.         pw.println(sdf2.format(d));
53.     }
54. }
```

- The file, named `IOCleanupUtils.java`, contains an overloaded `cleanup` method to handle each of the four main I/O data types.
  - Below is the method that takes a `Reader` as a parameter.

```
public static void cleanup(Reader reader){
    if(reader != null)
        try{
            reader.close();
        }catch(IOException ioe){
            System.err.println(MSG);
        }
}
```

Exercises

1. Modify a copy of the `Person` class from this chapter so that it has both a no args constructor and a constructor that takes a person's first and last name.

   - The `Person` should also have the requisite getter, setter, and `toString` methods.

2. Create a class named `Address` that represents a `street`, `city`, `state`, and `zipcode`.

   - Provide getter and setter methods for the private instance data.
   - The `toString` should return a two line `String` of the form:

     ```
     street
     city, state zip
     ```

   - Create an application to test the `Address` class.

3. Create a class named `Customer` that extends `Person`.

   - The `Customer` should be composed of an `Address` and a `double` representing the customer's credit balance.
     - The customer should also have a customer ID that is a six-digit `int` generated from a `static` variable of the class.
   - In addition to taking a first and last name, the constructor should be overloaded to allow either:
     - an `Address` to be passed as a parameter; or
     - a `street`, `city`, `state`, and `zip` as parameters.
   - Create an application to test the `Customer` class.

Chapter 2:
Packaging and Distributing a Java Application

1) Packages 2-2

2) Managing Source and Class Files 2-4

3) The `javadoc` Utility 2-10

4) Documenting Classes and Interfaces 2-12

5) Documenting Fields 2-13

6) Documenting Constructors and Methods 2-14

7) Running the `javadoc` Utility 2-16

8) `jar` Files 2-17

9) The Manifest File 2-19

10) Bundling and Using Jar-Packaged Resources 2-20

Packages

- A package in Java serves several purposes.
  - Packages provide a means of avoiding naming conflicts.
  - Packages represent collections of related classes, making data types easier to find and use.
  - Packages provide a way to control access to code.
- The classes, which are part of the Java platform, are members of various packages as shown below.
  - `java.lang` - fundamental classes
  - `java.util` - miscellaneous utility classes
  - `java.io` - classes for reading and writing (input and output)
  - `java.net` - classes related to networking
- The `package` statement is used to uniquely define a class.
  - A package is often referred to as the **namespace** of the class.
  - For example, if we were to develop a class named `String` (although not recommended), it would be necessary to distinguish between our `String` class and the one provided as part of the Java APIs.
- This section details the various issues in regards to developing and working with your own packages.

Packages

- Packages typically contain classes relating to one another.
  - For example, a `geometry` package might contain classes such as `Circle`, `Rectangle`, etc.
  - The classes in this course have taken this approach.
    - All of the source code provided has been divided into the four categories of **examples**, **exercises**, **starters,** and **solutions**.
    - Within each of these categories there are sub categories, such as the **examples** from this chapter being categorized as **packaging**.
    - There are also corresponding sub packages within **starters** and **solutions**.
  - The above convention results in package statements with names like the following.
    - `examples.review`
    - `examples.packaging`
    - `solutions.review`
    - `solutions.packaging`
- If a `package` statement is not used, the data type belongs to the unnamed (or default) package.
  - The unnamed package is often used when beginning the development process, but classes and interfaces really belong in named packages.

Managing Source and Class Files

- To understand how `.java` and `.class` files are managed, several data types will first be defined.

- The first data type, defined below, describes a type of vehicle and has been given a package name of `examples.packaging.vehicle`.

Car.java

```
 1.  package examples.packaging.vehicle;
 2.  public class Car {
 3.      private String make, model;
 4.      public Car(String make, String model) {
 5.          this.make = make;
 6.          this.model = model;
 7.      }
 8.      public String toString() {
 9.          return make + " " + model;
10.      }
11.  }
```

- One option available to a vehicle might be a DVD player.
  - Although `DVDPlayer` may be associated with a vehicle, it is not itself a vehicle. Therefore, it has been placed in a sub package named `options`.

DVDPlayer.java

```
 1.  package examples.packaging.vehicle.options;
 2.  public class DVDPlayer {
 3.      private String title;
 4.      public void insert(String title){
 5.          this.title = title;
 6.      }
 7.      public void play(){
 8.          System.out.println(title + " is playing.");
 9.      }
10.  }
```

Managing Source and Class Files

- An airplane can be thought of as a different type of vehicle and could be placed in the `examples.packaging.vehicle` package.
  - The airplane defined below utilizes a DVD player for in-flight entertainment.

Airplane.java

```
1.  package examples.packaging.vehicle;
2.  import examples.packaging.vehicle.options.DVDPlayer;
3.  public class Airplane {
4.      private DVDPlayer dvd = new DVDPlayer();
5.      public void startInFlightEntertainment(){
6.          dvd.insert("Airplane II");
7.          dvd.play();
8.      }
9.  }
```

  - Note that the `Airplane` class uses an `import` statement to import the `DVDPlayer` class.
    - This is used since the `DVDPlayer` class is in a different package than the `Airplane` class.
  - Another option would have been to rewrite the declaration of the `DVDPlayer` on line four in the code above.

```
examples.packaging.vehicle.options.DVDplayer dvd =
   new examples.packaging.vehicle.options.DVDplayer();
```

Managing Source and Class Files

- The class below tests the vehicle data types.

VehicleTesting.java

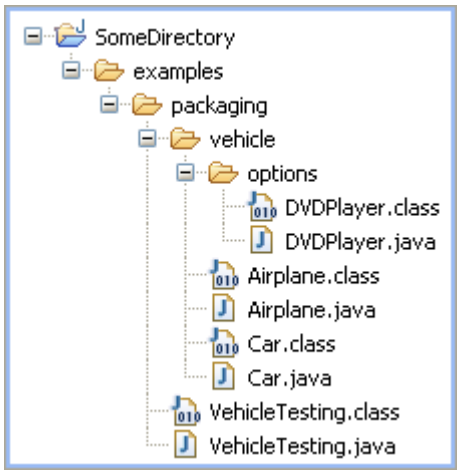
```
 1. package examples.packaging;
 2. import examples.packaging.vehicle.*;
 3. import examples.packaging.vehicle.options.*;
 4. public class VehicleTesting {
 5.     public static void main(String[] args) {
 6.         System.out.println(new Car("Honda", "Fit"));
 7.         Airplane a = new Airplane();
 8.         a.startInFlightEntertainment();
 9.         DVDPlayer player = new DVDPlayer();
10.         player.insert("Gone with the Wind");
11.         player.play();
12.     }
13. }
```

  - Note that import of `examples.packaging.vehicle.*` is used to import both the Airplane and Car classes but does not import the `DVDPlayer` since that class is in a different package.

Managing Source and Class Files

- The next several diagrams show directory structures in which the `.java` and `.class` files are typically stored.
  - Below each diagram is an indication of what would be typed on the command line to compile the `.java` files so that the `.class` files would be created in the directory structure shown.
  - It is important to realize that regardless of where the `.java` files are stored, the resultant `.class` files are ALWAYS required to be in a directory structure that matches the package names of the data types.

- An example of maintaining the source and class files in the same directory is shown below.



  - The `CLASSPATH` environment variable would be set to include "`SomeDirectory.`"
  - With the command line at the following directory:

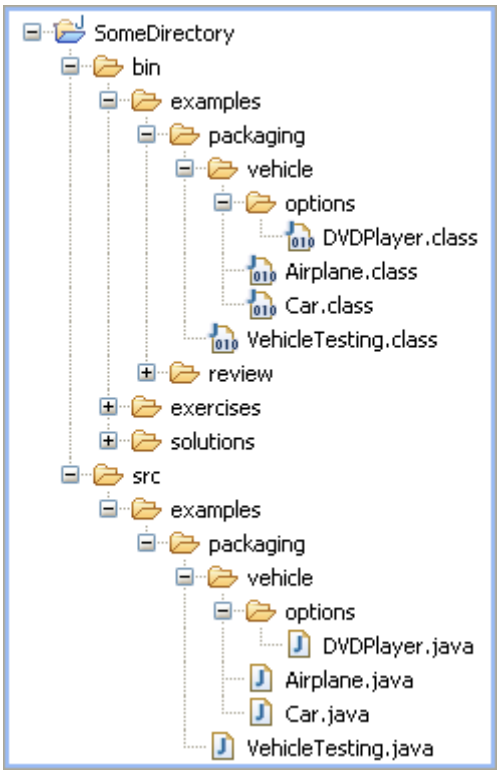
    `.../SomeDirectory/examples/packaging`

    - The following can be typed to compile `VehicleTesting.java` and its dependent data types.

      ```
      javac VehicleTesting.java
      ```

Managing Source and Class Files

- An example of maintaining the source and class files in separate directories is shown below.



  - The `CLASSPATH` environment variable would be set to include "`SomeDirectory/bin.`"
  - With the command line at the following directory:

    `.../SomeDirectory/src/examples/packaging`

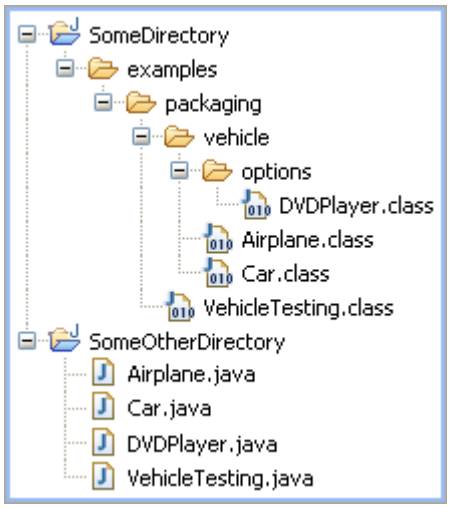
    - The following can be typed to compile `VehicleTesting.java` and its dependent data types.

      ```
      javac -d .../SomeDirectory/bin VehicleTesting.java
      ```

- The `-d` option to the java compiler is used to create the .class files in the directory (`. . ./SomeDirectory/bin`).
  - It automatically generates the necessary package structure for the resultant `.class` files if it does not exist.

Managing Source and Class Files

- An example of maintaining all the source files in a single directory can be seen below.



  - The `CLASSPATH` environment variable would be set to include "`SomeDirectory.`"
  - With the command line at the following directory:

    `.../SomeOtherDirectory`

    - The following can be typed to compile `VehicleTesting.java` and its dependent data types.

      ```
      javac -d .../SomeDirectory VehicleTesting.java
      ```

- Regardless of the directory structure chosen, the following command would be typed in order to run the example.

```
java examples.packaging.VehicleTesting
```

The `javadoc` Utility

- The `javadoc` utility is a tool for generating API documentation in HTML format from comments in source code.
  - A developer, through the use of document comments, can add custom information about the class to the generated API documentation.
  - The utility can be used to create documentation for entire packages and/or individual source files.
- Documentation comments are special comments that are delimited by "`/**`" and "`*/`" and placed in certain locations within the source code.
- The general syntax of a javadoc comment follows.

```
/**
 * Some short description that ends with a period. Often
 * followed by a more detailed description, that may
 * include some html tags such as <code>int</code><br>
 * <b>Info:</b>just wanted to show some bold text.
 *
 * Followed by some special tags depending on what it is
 * you are documenting.
 * @author /training/etc
 * @version 1.0
 */
```

  - Documentation comments can only be used effectively when they immediately precede interfaces, classes, constructors, fields, and/or methods.
  - Documentation comments placed in the body of a method are ignored.
  - Only one documentation comment per declaration statement is recognized by the Javadoc tool.

The `javadoc` Utility

- The special tags that can be used come in two types.
  - Block tags - These tags can be placed only in the tag section that follows the main description. Block tags are of the form `@tag`.
  - Inline tags - These can be placed anywhere in the main description or in the comments for block tags. Inline tags are denoted by curly braces. `{@tag}`
- The complete list of tags is shown below.

Javadoc Tags			
`@author`	`{@code}`	`{@docRoot}`	`@deprecated`
`@exception`	`{@inheritDoc}`	`{@link}`	`{@linkplain}`
`{@literal}`	`@param`	`@return`	`@see`
`@serial`	`@serialData`	`@serialField`	`@since`
`@throws`	`{@value}`	`@version`	

  - The list may vary slightly based on the version of Java.
- A detailed explanation of the tags and where they can be used can be found in the Java following documentation.

  `docs/technotes/tools/windows/javadoc.html`
- The examples that follow introduce some of the common tags by documenting the `Car` class defined earlier.
  - The examples pertaining to this portion of the materials can be found in the `examples.documenting` package.

Documenting Classes and Interfaces

- The example below is used to document the class itself.

Car.java

```
 1. package examples.documenting.vehicle;
 2. /**
 3.  * A Class representing a car. For Example:
 4.  * {@code Car c = new Car(make, model);}
 5.  *
 6.  * @author /training/etc
 7.  * @version 2.0, 7/1/2008
 8.  */
 9. public class Car {
10.  ...
11. }
```

- When documenting the class itself, it is important that the documentation comment immediately precede the class declaration.
  - A common mistake is to place the package and/or import statement(s) between the documentation comment and class declaration.
  - Any document comment that is not placed properly will be ignored by the `javadoc` tool.

- The following tags can be used when documenting classes and interfaces.

Class/Interface Tags		
@author	@since	@deprecated
@serial	@see	@version
{@link}	{@linkplain}	{@docRoot}

Documenting Fields

- The example below demonstrates adding documentation comments to the field(s) of a class.

Car.java

```
 1. package examples.packaging.vehicle;
 2. /** ... */
 3. public class Car {
 4.     /**
 5.      * The make and model of the Car. Such as
 6.      * "Ford" "Mustang".
 7.      */
 8.     private String make, model;
 9.  ...
10. }
```

- The Javadoc tool would generate documentation from the above code similar to:

```
private String make
     The make and model of the Car

private String model
     The make and model of the Car
```

  - For this reason, the single declaration for the `make` and `model` would be better handled as two separate declarations in the code.

- The following tags can be used when documenting fields.

Field Tags		
`@see`	`@since`	`@deprecated`
`@serial`	`@serialField`	`{@value}`
`{@link}`	`{@linkplain}`	`{@docRoot}`

Documenting Constructors and Methods

- The example below demonstrates adding documentation comments to the constructor(s) and method(s) of a class.

Car.java

```
 1. package examples.packaging.vehicle;
 2. /** ... */
 3. public class Car {
 4.     /** ... */
 5.     private String make, model;
 6.
 7.     /**
 8.      * Represents a car by its make and model.
 9.      * @param make The make of the car. <br>
10.      *     For Example: "Ford" or "Toyota"
11.      * @param model The model of the car. <br>
12.      *     For Example: "Taurus" or "Camry"
13.      */
14.     public Car(String make, String model) { ... }
15.
16.     /**
17.      * Returns the <code>String</code> representation
18.      * of a <code>Car</code>. In the form of:<br>
19.      * <code>make:model</code>
20.      * @return the make:model of the Car.
21.      * @see java.lang.Object#toString()
22.      * @since 1.0
23.      */
24.     public String toString() { ... }
25. }
```

- The following tags can be used when documenting constructors and methods.

Method/Constructor Tags		
@see	@since	@deprecated
@param	@return	@throws
@exception	@serialData	{@link}
{@linkplain}	{@inheritDoc}	{@docRoot}

Running the `javadoc` Utility

- In order to generate the actual documentation for the `Car` class, the following can be typed on the command line, from the directory that contains the file named `Car.java`.

```
javadoc -d %DOCS% Car.java
```

  - The `-d %DOCS%` option is used to specify the directory in which the documentation should be created.
    - Recall that the `DOCS` environment variable was set to the `docs` directory of the lab files in the `setenv` script.

- A long list of options can be specified when running the `javadoc` utility.
  - The complete list can be seen by entering `javadoc` on the command line.
  - Some options you may find useful are listed below.

Option	Explanation
`-d <directory>`	Destination directory for output files
`-version`	Include @version paragraphs
`-author`	Include @author paragraphs
`-public`	Show only public classes and members
`-protected`	Show protected/public (default)
`-package`	Show package/protected/public
`-private`	Show all classes and members
`-subpackages <pkglist>`	Specify subpackages to recursively load
`-classpath <pathlist>`	Specify where to find user class files
`-sourcepath <pathlist>`	Specify where to find source files

- The code on the following page demonstrates the command line to document the entire `examples.packaging` package and its sub-packages.

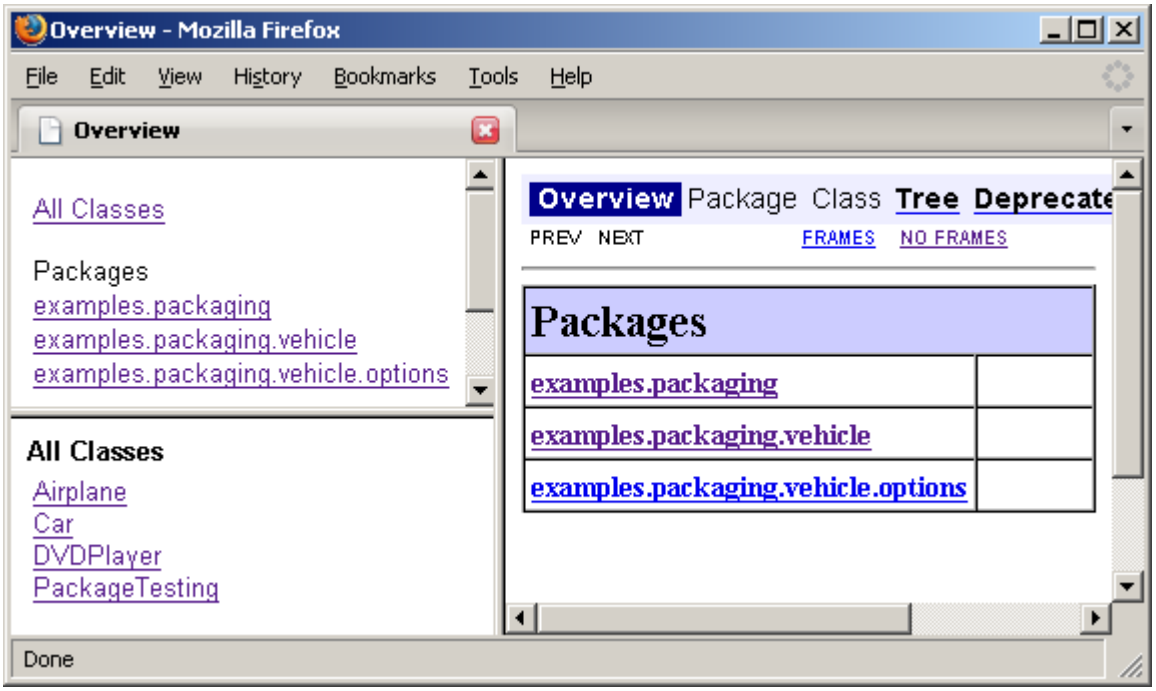
Running the `javadoc` Utility

- In order to document the entire `examples.packaging` package, the location of the `.java` source files will be specified using the `-sourcepath` option with the `javadoc` utility.

```
javadoc -sourcepath %SOURCE% -d %DOCS% -subpackages
examples.packaging
```

  - Note: The above two lines would be entered as a single line on the command line.

- Among all of the files produced by this command, you will see a file named `index.html` in the `docs` directory.



`jar` Files

- Many different utility programs maintain groups of files as a single archive.
  - In the UNIX world, the `ar` and `tar` utilities allow the creation, insertion, and extraction of files from a library.
  - In the Microsoft world, there are utilities that operate on `.zip` and `.cab` files and provide similar functionality.
  - In the Macintosh world, the utility programs create and manipulate `.sit` and `.dmg` files, to name a few.
- In the Java world, this concept has been expanded to the `jar` utility, a utility allowing the packaging of many files.
  - A typical `jar` file contains `.class` files and any other resources that the application relies upon.
  - A `jar` file can be named whatever you wish, but conventionally they have the `.jar` suffix.
    - `.war` (Web Applications) and `.ear` (Enterprise Applications) are other suffixes used in a J2EE environment.
- `jar` files have various uses in the Java world.
  - `.jar` files are often used to distribute a collection of `.class` files as a single archive.
    - Instead of having to expand the archive later, the `.class` files contained within the `.jar` file can be accessed simply by including the name of the `.jar` file in the `CLASSPATH`.
- The `jar` utility is explained in more detail on the next page.

The `jar` Utility

- The `jar` utility has the following options that can be specified when it is run.

Option	Description
`c`	Creates a new archive.
`t`	Lists a table of contents for archive.
`x`	Extracts the named (or all) files from archive.
`u`	Updates an existing archive.
`i`	Generates index information for specified jar file.
`v`	Generate verbose output to standard output.
`f`	Specifies the archive file name.
`m`	Includes manifest information from specified manifest file.
`0(zero)`	Only stores the information - does not use compression.
`M`	Does not create a manifest file for the entries
`C`	Changes to the specified directory and includes the file(s).

  - The `jar` utility recursively processes any directory.

- When you create a jar file, a manifest is automatically generated and included in the jar file.
  - The file is named `MANIFEST.MF` and is placed in a directory named `META_INF`.
  - The contents of the generated manifest look similar to the following.

```
Manifest-Version: 1.0
Created-By: 1.5.0_10 (Sun Microsystems Inc.)
```

    - The above values will depend on the version of Java.

  - The manifest file contains meta-information about the contents of the archive.
    - This information may further describe what is in the jar file.
    - This information is in the form of entries that contain key-value pairs (also referred to as headers and/or attributes).

The Manifest File

- The `jar` utility can also be used to bundle an application into an executable jar file.
  - In this case, information about launching the application must be added to the generated `MANIFEST.MF` file.
- The `m` option of the `jar` utility allows a file to be specified whose contents will be included in the generated manifest.
  - One such key-value pair is:

    `Main-Class:` *`fully.qualified.ClassName`*

    - The *`fully.qualified.ClassName`* represents the name of the class in the `.jar` file containing the `main` method that will be called by the JVM at startup time.
- The file below contains a key-value pair to be included in the manifest generated by the jar utility.

SampleManifest.txt

```
1.  Main-Class: examples.packaging.PackageTesting
2.
```

  - The blank line after the entry is necessary to demarcate the end of each entry.

Bundling and Using Jar-Packaged Resources

- To bundle the `VehicleTesting` application, it is necessary to ensure that the package structure of the class be replicated within the jar file itself.

- To ensure that the package structure is included as part of the jar process, run the `jar` utility from the directory that contains the root of the package structure.
  - In the case of `VehicleTesting`, the directory would be the one that has the `examples` directory in it.
    - The following command will bundle the contents of the `examples\packaging` directory into a file named `sample.jar`.

      ```
      jar cvf sample.jar examples/packaging/*
      ```

    - The following command will bundle the contents of the `examples\packaging` directory into an executable jar file named `sample2.jar`. (Note: The command should be typed on a single line.)

      ```
      jar cvmf
      examples/packaging/SampleManifest.txt
      sample2.jar examples/packaging/*
      ```

- The resultant jar file can now be executed from the command line by using the `-jar` option as shown below.

```
java -jar sample2.jar
```

Bundling and Using Jar-Packaged Resources

- A **resource** is data (text, images, etc.) that is accessible by the class code in a manner independent of the location of the code.
  - Suppose an application reads data from a text file.
    - If the application, along with the text file, is bundled into an executable jar file, there is a good chance that the application will no longer be able to read from the text file when the jar is executed.
    - This is because the code is probably not written to read from the jar file when necessary.
  - The application below will be used to demonstrate the potential problem.

ReadAFile.java

```
 1. package examples.resources.bad;
 2. import java.io.*;
 3. public class ReadAFile {
 4.     public static void main(String[] args)
 5.       throws IOException {
 6.         String fileName =
 7.             "bin/examples/resources/bad/data.txt";
 8.         File f = new File(fileName);
 9.         FileInputStream fis = new FileInputStream(f);
10.         int data;
11.         while((data = fis.read()) != -1){
12.             System.out.print((char) data);
13.         }
14.         fis.close();
15.         System.out.println();
16.         FileReader fr = new FileReader(f);
17.         while((data = fr.read()) != -1){
18.             System.out.print((char) data);
19.         }
20.         fr.close();
21.     }
22. }
```

Bundling and Using Jar-Packaged Resources

- Running the previous code should result in the `FileInputStream` and `FileReader` successfully reading the contents of the text file.
  - It is when the code and the text file are bundled into a jar file and executed that it may not be able to locate the `data.txt` file.
  - In order to avoid the above situation, it is often desired to reference the file as a resource within your code such that its location is independent of the location of the code.
    - This is accomplished by referencing the file relative to a class file's location.
- Instances of the class `Class` are used in a running Java application to represent each class file loaded into memory.
  - Every object has a `getClass()` method that can be called on an instance of the object to get a reference to its `Class` object.
  - Every object also has a static property named `class` that is a reference to its `Class` object.
- Once a reference to the class has been obtained, the following method can be used to access a resource that an application may need.

```
public InputStream getResourceAsStream(String name)
```

  - The resource will automatically be loaded from a path relative to the `.class` file.
    - If the `.class` file was loaded from a jar file, then the resource will be retrieved from the jar file.

Bundling and Using Jar-Packaged Resources

- The example below is a rewrite of the previous example.
  - The application demonstrates the `getResourceAsStream` and `getResource` methods to read the contents of a resource associated with the application.

ReadAFile.java

```
 1.  package examples.resources.good;
 2.  import java.io.*;
 3.  public class ReadAFile {
 4.      public static void main(String[] args)
 5.        throws Exception {
 6.          Class c = ReadAFile.class;
 7.          InputStream fis =
 8.              c.getResourceAsStream("data.txt");
 9.          int data;
10.          while((data = fis.read()) != -1){
11.              System.out.print((char) data);
12.          }
13.          fis.close();
14.          System.out.println();
15.          fis = c.getResourceAsStream("data.txt");
16.          InputStreamReader isr =
17.              new InputStreamReader(fis);
18.          BufferedReader br = new BufferedReader(isr);
19.          String txt;
20.          while((txt = br.readLine()) != null){
21.              System.out.println(txt);
22.          }
23.          br.close();
24.      }
25.  }
```

- The above version of the application will successfully read from the text file even when the application is bundled as an executable jar.

Exercises

1. This exercise should be started with a copy of the `Address`, `Person`, `Customer` classes created earlier as an exercise.
   - If the classes were not created earlier, copies of these classes are available in the `starters` directory of this chapter.
   - Use `javadoc` style comments to document at least one field, method, and constructor from each of the classes.
2. Use the `javadoc` utility to generate the documentation, from Exercise 1 above, into a directory of your choosing.
3. Use the `jar` utility to bundle the docs created above into a single file to make it easier to distibute the documentation for your classes.
4. Use the `jar` utility to bundle the class files from above into an executable jar file.
   - This will require the inclusion of a copy of the application written earlier to test the `Customer` class.
     - Once again, if the test class was not created earlier, one is supplied in the starters directory of this chapter.

Chapter 3:
Miscellaneous Enhancements

1) Enhanced `for` Loop 3-2

2) Autoboxing and Auto-Unboxing 3-4

3) Static Imports 3-6

4) `varArgs` 3-8

5) Typesafe Enums 3-12

6) Formatted Strings 3-16

7) Format Specifier Syntax 3-17

8) Format Specifier Conversions 3-18

9) Format Specifier Flags 3-22

10) Formatted Integers Example 3-23

11) Formatted Floating Points Example 3-24

12) Formatted Strings Example 3-25

13) Formatted Dates Example 3-26

14) Complex Formatted Example 3-27

Enhanced `for` Loop

- The enhanced `for` loop is a Java language construct introduced in Java SE 5.
  - The enhanced `for` loop eliminates the error-proneness of index variables and iterators when iterating over arrays and collections.
  - The enhanced `for` loop provides the ability to access the content of arrays and collections without the need for an index or an iterator.
  - The general syntax for the enhanced `for` loop follows.

    ```
    for(Type Identifier : Expression)
    ```

  - The example below uses an enhanced `for` loop to calculate the sum of the values in an `int` array.

ForEachWithArray.java

```
 1.  package examples.enhancements;
 2.
 3.  public class ForEachWithArray {
 4.      public static void main(String args[]){
 5.          int a [] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
 6.          int sum = 0;
 7.          for(int i:a)
 8.              sum += i;
 9.          System.out.println(sum);
10.      }
11.  }
```

  - The colon (`:`) in the enhanced for loop is read as "in."
  - The loop above is read as "for each `int i` in `a`."

- An example of using an enhanced for loop to iterate through a `Vector` is shown on the next page.

Enhanced `for` Loop

- The example below demonstrates two techniques for iterating through a `Properties` object.
  - The enhanced for loop, used in the `newForLoop` method call, removes the need for an `Enumeration` and a lot of the clutter found in the `oldForLoop` method call.

ForEachWithProperties.java

```
 1. package examples.enhancements;
 2.
 3. import java.util.Enumeration;
 4. import java.util.Properties;
 5.
 6. public class ForEachWithProperties {
 7.     public static void main(String args[]){
 8.         Properties sysProps = System.getProperties();
 9.         oldForLoop(sysProps);
10.         System.out.println("\n\n--------------\n\n");
11.         newForLoop(sysProps);
12.     }
13.
14.     private static void oldForLoop(Properties p){
15.         Enumeration e;
16.         for(e= p.elements(); e.hasMoreElements();)
17.             System.out.println(e.nextElement());
18.     }
19.
20.     private static void newForLoop(Properties p){
21.         for(Object value : p.values())
22.             System.out.println(value);
23.     }
24. }
```

Autoboxing and Auto-Unboxing

- Since a specific data type, such as `Vector`, can only hold object references, the wrapper classes convert primitive types like `int` and `double` into their equivalent `Object`-based counterparts like `Integer` and `Double`.
  - Primitives are "boxed" into their appropriate wrapper class via one of the constructors of the wrapper class.
  - Primitives are "unboxed" through method calls on the object such as the `intValue` method from the `Integer` class.
- The **autoboxing** and **auto-unboxing** (introduced in Java SE 5) of Java primitives produces code that is more concise and easy to follow.
  - The conversion required to transition a primitive to its equivalent `Object`-based counterpart and back is left to the compiler.
  - Autoboxing and auto-unboxing tend to blur the distinction between primitive and reference types but does not eliminate it.
    - This new facility is often used when a numerical value needs to be put into a `Collection`.
    - It would not be appropriate to use this new facility for performance-sensitive numerical code.
- The example on the following page demonstrates the use of both the older style of code and the new autoboxing facility.

Autoboxing and Auto-Unboxing

AutoBoxing.java

```
 1. package examples.enhancements;
 2. import java.util.ArrayList;
 3. import java.util.Vector;
 4.
 5. public class AutoBoxing {
 6.
 7.     public static void main(String[] args) {
 8.         int numbers [] ={1 , 2, 3, 4, 5, 6, 7, 8, 9};
 9.         useWrapperClasses(numbers);
10.         useAutoBoxing(numbers);
11.     }
12.
13.     private static void useWrapperClasses(int n []){
14.         Vector v = new Vector();
15.         // populate list with numbers
16.         for(int i = 0; i < n.length; i++)
17.             v.add(new Integer(n[i]));
18.         //calculate sum of numbers in the list
19.         int sum = 0;
20.         for(int i = 0; i < v.size(); i++)
21.             sum += ((Integer) v.get(i)).intValue();
22.         System.out.println(sum);
23.     }
24.
25.     private static void useAutoBoxing(int n []){
26.         Vector v = new Vector();
27.         // populate list with numbers
28.         for(int i = 0; i < n.length; i++)
29.             v.add(n[i]);
30.         //calculate sum of numbers in the list
31.         int sum = 0;
32.         int temp;
33.         for(int i = 0; i < v.size(); i++){
34.             temp = (Integer) v.get(i);
35.             sum += temp;
36.         }
37.         System.out.println(sum);
38.     }
39. }
```

Static Imports

- In order to access static members of a class, it is necessary to qualify references with the class from which they came.
  - This can be seen in the code below, where access to the static members `ceil`, `random`, `pow`, and `PI` are needed from the `Math` class.

AccessStaticMembers1.java

```
 1. package examples.enhancements;
 2.
 3. public class AccessStaticMembers1 {
 4.
 5.     public static void main(String[] args) {
 6.         //Generate a random number between 1 and 10
 7.         int x = (int) Math.ceil(Math.random() * 10);
 8.         System.out.println("Random Number:" + x);
 9.
10.         //Calculate area of circle with radius of 4
11.         int radius = 4;
12.         double area = Math.PI * Math.pow(radius, 2);
13.         System.out.println("Area:" + area);
14.     }
15. }
```

- A **static import** (introduced in Java SE 5) allows unqualified access to static members without inheriting from the class containing the static members.
  - Instead, the program imports the static members.
  - Once the static members have been imported, they may be used without qualification.
- The example on the next page is a rewrite of the above program, using static imports.

Static Imports

AccessStaticMembers2.java

```
 1. package examples.enhancements;
 2. import static java.lang.Math.ceil;
 3. import static java.lang.Math.random;
 4. import static java.lang.Math.pow;
 5. import static java.lang.Math.PI;
 6.
 7. public class AccessStaticMembers2 {
 8.
 9.     public static void main(String[] args) {
10.         //Generate a random number between 1 and 10
11.         int x = (int) ceil(random() * 10);
12.         System.out.println("Random Number:" + x);
13.
14.         //Calculate area of circle with radius of 4
15.         int radius = 4;
16.         double area = PI * pow(radius, 2);
17.         System.out.println("Area:" + area);
18.     }
19. }
```

- While static imports can make your program more readable by removing the repetition of class names, importing all of the static members from a class can be particularly harmful to readability.
  - An example of this would be when you import static members from more than one class, such as:

```
import static java.lang.Math.*;
import static java.util.Arrays.*;
```

    - It would become difficult for a reader to determine if a static member in the code comes from the `Math` class or the `Arrays` class.

varArgs

- Prior to Java SE 5, a method taking an arbitrary number of values required the creation of an array and the placement of the values into the array prior to invoking the method.
  - To demonstrate this, the example below defines a simple date as containing a month, day, and year.
    - The class contains a private method named `helper` that takes an `int` array as a parameter.
    - The method is designed to be used by all of the constructors of the class by allowing a variable number of parameters to be specified indicating the `month`, `day`, and `year`.

SimpleDate1.java

```
 1.  package examples.enhancements;
 2.  import java.util.*;
 3.
 4.  public class SimpleDate1 {
 5.      int month, day, year;
 6.      private void helper(int varArgs []){
 7.          Calendar now = new GregorianCalendar();
 8.          month = now.get(Calendar.MONTH) + 1;
 9.          day = now.get(Calendar.DAY_OF_MONTH);
10.          year = now.get(Calendar.YEAR);
11.          switch(varArgs.length){
12.              case 1:
13.                  day   = varArgs[0];
14.                  break;
15.              case 2:
16.                  month = varArgs[0];
17.                  day   = varArgs[1];
18.                  break;
19.              case 3:
20.                  month = varArgs[0];
21.                  day   = varArgs[1];
22.                  year  = varArgs[2];
23.                  break;
24.          }
25.      }
```

varArgs

SimpleDate1.java *continued*

```
26.      public SimpleDate1(){
27.          helper(new int [0]);
28.      }
29.      public SimpleDate1(int d){
30.          int x [] = {d};
31.          helper(x);
32.      }
33.      public SimpleDate1(int m, int d){
34.          int x [] = {m, d};
35.          helper(x);
36.      }
37.      public SimpleDate1(int m , int d, int y){
38.          int x [] = {m, d, y};
39.          helper(x);
40.      }
41.      public String toString(){
42.          return month + "/" + day + "/" + year;
43.      }
44.  }
```

- Java SE 5 introduced a `varArgs` feature allowing multiple arguments to be passed as parameters to a single method declaration.
  - It requires the simple ... notation for the method that accepts the argument list.
  - While it is still true that multiple arguments must be passed in an array, the `varArgs` feature automates and hides the process.
- The example on the next page is a simplified version of the `SimpleDate1` class, which utilizes the `varArgs` feature.
  - Note that `SimpleDate1.java` used four constructors and the `helper` method, whereas `SimpleDate2.java` has only one constructor and has no need for the `helper` method.

varArgs

SimpleDate2.java

```
 1. package examples.enhancements;
 2. import java.util.*;
 3.
 4. public class SimpleDate2 {
 5.     int month, day, year;
 6.
 7.     public SimpleDate2(int ... varArgs){
 8.         Calendar now = new GregorianCalendar();
 9.         month = now.get(Calendar.MONTH) + 1;
10.         day = now.get(Calendar.DAY_OF_MONTH);
11.         year = now.get(Calendar.YEAR);
12.         switch(varArgs.length){
13.             case 1:
14.                 day   = varArgs[0];
15.                 break;
16.             case 2:
17.                 month = varArgs[0];
18.                 day   = varArgs[1];
19.                 break;
20.             case 3:
21.                 month = varArgs[0];
22.                 day   = varArgs[1];
23.                 year  = varArgs[2];
24.                 break;
25.         }
26.     }
27.
28.     public String toString(){
29.         return month + "/" + day + "/" + year;
30.     }
31. }
```

- A program that tests both `SimpleDate1` and `SimpleDate2` is shown on the following page.

varArgs

SimpleDateTester.java

```
 1. package examples.enhancements;
 2.
 3. public class SimpleDateTester {
 4.
 5.     public static void main(String[] args) {
 6.         print(new SimpleDate1());
 7.         print(new SimpleDate1(17));
 8.         print(new SimpleDate1(3, 17));
 9.         print(new SimpleDate1(3,17,2007));
10.         print("--------------------------------");
11.         print(new SimpleDate2());
12.         print(new SimpleDate2(17));
13.         print(new SimpleDate2(3, 17));
14.         print(new SimpleDate2(3,17,2007));
15.     }
16.
17.     private static void print(Object o){
18.         System.out.println(o);
19.     }
20. }
```

Typesafe Enums

- In Java releases prior to Java SE 5, the standard way to represent an enumerated type was the `int enum` pattern.

```
// int Enum Pattern - has severe problems!
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

- This pattern has problems, and some are listed below.
  - Not type-safe
    - Since a season is just an `int`, you can pass in any other *`int`* value where a season is required or add seasons together (which makes no sense).
  - Printed values are uninformative
    - Because they are `int`s, if printed all you get is a number, which tells you nothing about what it represents or even its type.
- Support for enumerated types was introduced in Java SE 5.
  - In their simplest form, these `enum`s take the following form.

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

- The new `enum` declaration defines a full-fledged class.
  - `enum` types have additional features.
    - Arbitrary methods and fields can be added to them.
    - They are `Comparable` and `Serializable`.
    - They are able to implement other interfaces.

Typesafe Enums

- The example below is a simple application that utilizes two `enum` types also declared in the example.

SimpleEnums.java

```
 1. package examples.enhancements;
 2.
 3. public class SimpleEnums{
 4.     enum MonthEnum {JANUARY, FEBRUARY, MARCH,
 5.         APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
 6.         OCTOBER, NOVEMBER, DECEMBER}
 7.
 8.     enum SeasonEnum {SPRING, SUMMER, FALL, WINTER}
 9.
10.     public static void main(String args[]){
11.         MonthEnum m = MonthEnum.JANUARY;
12.         SeasonEnum s = getSeason(m);
13.         System.out.println(m + " is in the " + s);
14.     }
15.     private static SeasonEnum getSeason(MonthEnum m){
16.         switch(m){
17.             case MARCH:
18.             case APRIL:
19.             case MAY:
20.          return SeasonEnum.SPRING;
21.             case JUNE:
22.             case JULY:
23.             case AUGUST:
24.          return SeasonEnum.SUMMER;
25.             case SEPTEMBER:
26.             case OCTOBER:
27.             case NOVEMBER:
28.          return SeasonEnum.FALL;
29.             case DECEMBER:
30.             case JANUARY:
31.             case FEBRUARY:
32.          return SeasonEnum.WINTER;
33.             default:
34.          return null;
35.         }
36.     }
37. }
```

Typesafe Enums

- The example below provides additional information to an `enum` type by including methods and fields in the declaration.

Month.java

```
 1.  package examples.enhancements;
 2.
 3.  public enum Month {
 4.      JANUARY(31), FEBRUARY(28), MARCH(31), APRIL(30),
 5.      MAY(31), JUNE(30), JULY(31), AUGUST(31),
 6.      SEPTEMBER(30), OCTOBER(31), NOVEMBER(30),
 7.      DECEMBER(31);
 8.
 9.      private final int days; // days in month
10.
11.      Month (int days){
12.          this.days = days;
13.      }
14.
15.      int getDays(){
16.          return days;
17.      }
18.  }
```

- The compiler automatically adds a static `values` method when it creates an `enum`.
  - The static `values` method returns an array containing all of the values of the `enum` in the order they are declared.
  - This method is commonly used in combination with for loops to iterate over the values of an `enum` type.

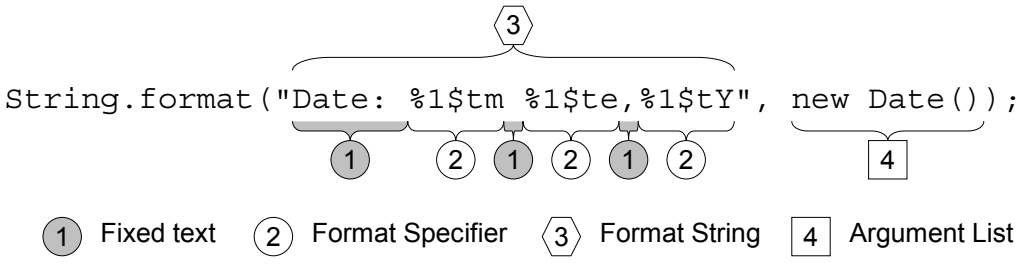
```
for (Month m : Month.values()) {
    System.out.print("The Month is " + m);
}
```

Typesafe Enums

- `enum` types have the following provisions associated with their use.
  - All `enums` implicitly extend `java.lang.Enum`.
    - Since Java does not support multiple inheritances, an `enum` cannot extend anything else.
  - Values are passed to the constructor when the constant is created.
  - Java requires that the constants be defined first, prior to any fields or methods.
    - When fields and methods exist, the list of `enum` constants must end with a semicolon.
  - The constructor for an `enum` type must be package-private or private access.
  - You cannot invoke an `enum` constructor yourself.

Formatted Strings

- The `java.util.Formatter` class (introduced in Java SE 5) brings `printf`-style format strings to the Java language.
  - This class provides layout justification and alignment, common formats for numeric, string, and date/time data.
  - The format strings are intended to be recognizable to C programmers but not necessarily completely compatible with those in C.
- Many of the existing classes in the Java SE have been updated to include `format()` and/or `printf()` convenience methods rather than requiring the end user to deal with the `Formatter` class directly.
  - Every method that produces formatted output requires a format string and an argument list.
    - The format string is a `String` which may contain fixed text and one or more embedded format specifiers.
    - The argument list consists of all arguments passed to the method after the format string. This argument list is built around the `varArgs` feature discussed earlier.
    - The diagram below breaks down the various pieces of the formatted output.



Format Specifier Syntax

- The format specifiers for general, character, and numeric types have the following syntax.

```
%[argument_index$][flags][width][.precision]conversion
```

  - The optional `argument_index$` is the position of the argument in the argument list.
    - The first argument is referenced by "`1$`," and the second is referenced by "`2$`," etc.
  - The optional `flags` is a set of characters that modify the output format.
    - The set of valid flags depends on the conversion.
  - The optional `width` is a non-negative integer indicating the minimum number of characters to be written to the output.
  - The optional `precision` is a non-negative integer usually used to restrict the number of characters.
    - The specific behavior depends on the conversion.
  - The required `conversion` is a character indicating how the argument should be formatted.
    - The set of valid conversions for a given argument depends on the argument's data type.

- The format specifiers for dates and times have the syntax:

```
%[argument_index$][flags][width]conversion
```

  - The required conversion is a two-character sequence.
    - The first character is '`t`' or '`T`.'
    - The second character indicates the format to use.

Format Specifier Conversions

- The last part of a format specifier, the conversion, is divided into the following categories.
  - **General**
    - May be applied to any argument type
  - **Character**
    - May be applied to basic types which represent Unicode characters: `char`, `Character`, `byte`, `Byte`, `short`, and `Short`
  - **Numeric**
    - Integral - may be applied to Java integral types: `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, and `BigInteger`
    - Floating Point - may be applied to Java floating-point types: `float`, `Float`, `double`, `Double`, and `BigDecimal`
  - **Date/Time**
    - May be applied to Java types which are capable of encoding a date or time: `long`, `Long`, `Calendar`, and `Date`
  - **Percent**
    - Produces a literal '`%`' ('\u0025')
  - **Line Separator**
    - Produces the platform-specific line separator

Format Specifier Conversions

- The following tables describe the available format specifier conversions for each category.

General Conversions	
Conv.	**Description**
'b' 'B'	If the argument `arg` is `null`, then the result is "`false`." If `arg` is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf()`. Otherwise, the result is "`true`."
'h' 'H'	If the argument `arg` is `null`, then the result is "`null`." Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.
's' 'S'	If the argument `arg` is `null`, then the result is "`null`." If `arg` implements `Formattable`, then `arg.formatTo` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.

Character Conversions	
Conv.	**Description**
'c' 'C'	The result is a Unicode character.

Integral Conversions	
Conv.	**Description**
'd'	The result is formatted as a decimal integer.
'o'	The result is formatted as an octal integer.
'x' 'X'	The result is formatted as a hexadecimal integer.

Floating Point Conversions	
Conv.	**Description**
'e' 'E'	The result is formatted as a decimal number in computerized scientific notation.
'f'	The result is formatted as a decimal number.
'g' 'G'	The result is formatted using computerized scientific notation or decimal format, depending on the precision and the value after rounding.
'a' 'A'	The result is formatted as a hexadecimal floating-point number with a significand and an exponent.

Format Specifier Conversions

Percent and Line Separator Conversions	
Conv.	**Description**
'%'	The result is a literal '%' ('\u0025').
'n'	The result is the platform-specific line separator.

Time Conversions	
Conv.	**Description**
'tH' 'TH'	Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. 00 - 23.
'tI' 'TI'	Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. 01 - 12.
'tk' 'Tk'	Hour of the day for the 24-hour clock, i.e. 0 - 23.
'tl' 'Tl'	Hour for the 12-hour clock, i.e. 1 - 12.
'tM' 'TM'	Minute within the hour formatted as two digits with a leading zero as necessary, i.e. 00 - 59.
'tS' 'TS'	Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. 00 - 60 ("60" is to support leap seconds).
'tL' 'TL'	Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. 000 - 999.
'tN' 'TN'	Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. 000000000 - 999999999.
'tp' 'Tp'	Locale-specific morning or afternoon marker in lower case, e.g."am" or "pm". Use of the conversion prefix 'T' forces this output to upper case.
'tz' 'Tz'	RFC 822 style numeric time zone offset from GMT, e.g. -0800.
'tZ' 'TZ'	A string representing the abbreviation for the time zone. The Formatter's locale will supersede the locale of the argument (if any).
'ts' 'Ts'	Seconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE/1000 to Long.MAX_VALUE/1000.
'tQ' 'TQ'	Milliseconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE to Long.MAX_VALUE.

Format Specifier Conversions

Date Conversions	
Conv.	**Description**
'tB' 'TB'	Locale-specific full month name, e.g. "January", "February."
'tb' 'th' 'Tb' 'Th'	Locale-specific abbreviated month name, e.g. "Jan", "Feb."
'tA' 'TA'	Locale-specific full name of the day of the week, e.g. "Sunday." "Monday"
'ta' 'Ta'	Locale-specific short name of the day of the week, e.g. "Sun", "Mon"
'tC' 'TC'	Four-digit year divided by 100, formatted as two digits with leading zero as necessary, i.e. 00 - 99
'tY' 'TY'	Year, formatted as at least four digits with leading zeros as necessary, e.g. 0092 equals 92 CE for the Gregorian calendar
'ty' 'Ty'	Last two digits of the year, formatted with leading zeros as necessary, i.e. 00 - 99
'tj' 'Tj'	Day of year, formatted as three digits with leading zeros as necessary, e.g. 001 - 366 for the Gregorian calendar
'tm' 'Tm'	Month, formatted as two digits with leading zeros as necessary, i.e. 01 - 13
'td' 'Td'	Day of month, formatted as two digits with leading zeros as necessary, i.e. 01 - 31
'te' 'Te'	Day of month, formatted as two digits, i.e. 1 - 31

Date/Time Composition Conversions	
Conv.	**Description**
'tR' 'TR'	Time formatted for the 24-hour clock as "%tH:%tM"
'tT' 'TT'	Time formatted for the 24-hour clock as "%tH:%tM:%tS"
'tr' 'Tr'	Time formatted for the 12-hour clock as "%tI:%tM:%tS %Tp"
'tD' 'TD'	Date formatted as "%tm/%td/%ty"
'tF' 'TF'	ISO 8601 complete date formatted as "%tY-%tm-%td"
'tc' 'TC'	Date and time formatted as "%ta %tb %td %tT %tZ %tY", e.g. "Sun Jul 20 16:17:00 EDT 1969"

Format Specifier Flags

- The table below describes the available format specifier flags and the category(s) to which they are applicable.

Format Specifier Flags						
Flag	Category					Description
	General	Character	Integral	Floating Point	Date/Time	
'-'	(1)	(1)	(1)	(1)	(1)	The result will be left-justified.
'#'	(2)		(4)	(1)		The result should use a conversion-dependent alternate form.
'+'			(5)	(1)		The result will always include a sign.
' '			(5)	(1)		The result will include a leading space for positive values.
'0'			(1)	(1)		The result will be zero-padded.
','			(3)	(6)		The result will include locale-specific grouping separators.
'('			(5)	(6)		The result will enclose negative numbers in parentheses.

(1) Flag is supported without restrictions
(2) Depends on the definition of `Formattable`
(3) For '`d`' conversion only
(4) For '`o`', '`x`', and '`X`' conversions only
(5) For '`d`', '`o`', '`x`', and '`X`' conversions applied to `BigInteger` or '`d`' applied to `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, and `Long`
(6) For '`e`', '`E`', '`f`', '`g`', and '`G`' conversions only

Formatted Integers Example

- The examples that follow demonstrate a variety of formatted strings.
  - In order to better visualize whitespace and padding within the output, each example's output is displayed within a grid.
    - The grid itself is not part of the output produced by the program.

FormattedIntegers.java

```
 1. package examples.enhancements;
 2.
 3. public class FormattedIntegers {
 4.     public static void main(String args[]){
 5.         int x = 456;
 6.         printf(" 1:" + "%d",x); // Decimal
 7.         printf(" 2:" + "%o",x); // Octal
 8.         printf(" 3:" + "%x",x); // Hex (lower)
 9.         printf(" 4:" + "%X",x); // Hex (upper)
10.         printf(" 5:" + "%10d",x);
11.         printf(" 6:" + "%-10d",x);
12.         printf(" 7:" + "%010d",x);
13.         printf(" 8:" + "%(d",-x);
14.         printf(" 9:" + "%+d",x);
15.         printf("10:" + "%+d",-x);
16.     }
17.     public static void printf(String f, Object ...a){
18.         System.out.printf(f + "%n", a);
19.     }
20. }
```

```
 1:456
 2:710
 3:1c8
 4:1C8
 5:       456
 6:456
 7:0000000456
 8:(456)
 9:+456
10:-456
```

Formatted Floating Points Example

FormattedFloatingPoints.java

```
 1. package examples.enhancements;
 2.
 3. public class FormattedFloatingPoints {
 4.     public static void main(String args[]){
 5.         double x = 1234567.0 + 2.0/3.0;
 6.         printf(" 1:" + "%f",x);
 7.         printf(" 2:" + "%.9f",x);
 8.         printf(" 3:" + "%.3f",x);
 9.         printf(" 4:" + "%15.3f",x);
10.         printf(" 5:" + "%015.3f",x);
11.         printf(" 6:" + "%,f",x);
12.         printf(" 7:" + "%e",x);
13.         printf(" 8:" + "%.10e",x);
14.         printf(" 9:" + "%g",x);
15.         printf("10:" + "%.9g",x);
16.     }
17.     public static void printf(String f, Object ...a){
18.         System.out.printf(f + "%n", a);
19.     }
20. }
```

```
 1:1234567.666667
 2:1234567.666666667
 3:1234567.667
 4:      1234567.667
 5:0001234567.667
 6:1,234,567.666667
 7:1.234568e+06
 8:1.2345676667e+06
 9:1.23457e+06
10:1234567.67
```

Formatted Strings Example

FormattedStrings.java

```
 1. package examples.enhancements;
 2.
 3. public class FormattedStrings {
 4.     public static void main(String args[]){
 5.         String text [] = {
 6.                 "Goodbye",
 7.                 "Hello",
 8.                 "A String of 25 characters",
 9.                 "goodbye",
10.                 "hello",
11.         };
12.         printf("%s",text);
13.         printf("%S",text);
14.         printf("%.4S",text);
15.         System.out.println("----------------------");
16.         for(String t:text){
17.             printf("%30s", t);
18.         }
19.
20.     }
21.     public static void printf(String f, Object ...a){
22.         System.out.printf(f + "%n", a);
23.     }
24. }
```

```
Goodbye
GOODBYE
GOOD
------------------------
                       Goodbye
                         Hello
             A String of 25 characters
                       goodbye
                         hello
```

Formatted Dates Example

FormattedDates.java

```
 1. package examples.enhancements;
 2.
 3. import java.util.Calendar;
 4. import java.util.GregorianCalendar;
 5.
 6. public class FormattedDates {
 7.     public static void main(String args[]){
 8.         // February 9, 2008 23:45:01
 9.         Calendar c =
10.             new GregorianCalendar(2008,1,9,23,45,01);
11.         printf("%1$tb or %1$tB or %1$tm", c);
12.         printf("%1$ta or %1$tA", c);
13.         printf("%1$tH:%1$tM:%1$tS", c);
14.         printf("%1$tI:%1$tM:%1$tS %1$tp", c);
15.         printf("%tR", c);
16.         printf("%tT", c);
17.         printf("%tr", c);
18.         printf("%tD", c);
19.         printf("%tc", c);
20.     }
21.     public static void printf(String f, Object ...a){
22.         System.out.printf(f + "%n", a);
23.     }
24. }
```

```
Feb or February or 02
Sat or Saturday
23:45:01
11:45:01 pm
23:45
23:45:01
11:45:01 PM
02/09/08
Sat Feb 09 23:45:01 EST 2008
```

Complex Formatted Example

- The following application combines many of the format specifiers used in the previous examples.
  - The application will consist of a list of employees that, when printed, should appear in columnar format.

Employee.java

```
 1.  package examples.enhancements;
 2.
 3.  import java.util.Calendar;
 4.
 5.  public class Employee {
 6.      private String firstName, lastName;
 7.      private Calendar hireDate;
 8.      private int id;
 9.      private static int idGenerator = 900;
10.      public Employee(String firstName,
11.              String lastName, Calendar hireDate){
12.          this.firstName = firstName;
13.          this.lastName = lastName;
14.          this.id = idGenerator += 100;
15.      }
16.      public String getFirstName() {
17.          return firstName;
18.      }
19.      public Calendar getHireDate() {
20.          return hireDate;
21.      }
22.      public int getId() {
23.          return id;
24.      }
25.      public String getLastName() {
26.          return lastName;
27.      }
28.  }
```

- The application that uses the `Employee` class above is shown on the following page.

Complex Formatted Example

ComplexFormatting.java

```
 1. package examples.enhancements;
 2.
 3. import java.util.GregorianCalendar;
 4.
 5. public class ComplexFormatting {
 6.     public static void main(String args[]){
 7.         Employee employee [] = new Employee[3];
 8.         employee[0] = new Employee("Joe", "Smith",
 9.                 new GregorianCalendar(2008,0,1));
10.         employee[1] = new Employee("Michelle",
11.                 "McDonald",
12.                 new GregorianCalendar(2008,9,10));
13.         employee[2] = new Employee("Peter", "Jones",
14.                 new GregorianCalendar(2008,4,17));
15.         for(Employee e:employee){
16.             printf("%1$-10s|%2$-10s|%3$5d|%4$tD" ,
17.                     e.getFirstName(),
18.                     e.getLastName(),
19.                     e.getId(),
20.                     e.getHireDate());
21.         }
22.     }
23.     public static void printf(String f, Object ...a){
24.         System.out.printf(f + "%n", a);
25.     }
26. }
```

```
Joe       |Smith     |  900|01/01/08
Michelle  |McDonald  | 1000|10/10/08
Peter     |Jones     | 1100|05/17/08
```

Exercises

1. Write and test a utility method that calculates and returns the average of a variable number of `int` arguments.
   - The use of an enhanced for loop should be used to loop through the arguments.
2. Add another utility method that calculates and returns the average of a variable number of `double` arguments.
3. Overload the method that takes a variable number of doubles as arguments with the following behavior.
   - The method should take, as its first parameter, an `int` indicating the number of places past the decimal that should be used to format the result.
   - The method should return a `String` representing the formatted result.
   - Note that although the method can be overloaded - it cannot be called due to the ambiguous method declarations.
     - To get both methods to work, they will actually have to be named differently.
4. Rewrite the `Address` class from the earlier exercises so that it allows a user to specify an address type (such as "Home" or "Work."
   - The address type should be defined an `Enum`.
   - The name of the new `Address` class should be `EnumeratedAddress` to differentiate it from the original version.

This Page Intentionally Left Blank

Chapter 4:
Assertions

1) Introduction 4-2

2) Assertion Syntax 4-3

3) Compiling with Assertions 4-5

4) Enabling and Disabling Assertions 4-6

5) Assertion Usage 4-9

Introduction

- Assertions provide programmers with a facility to test for conditions that they assume are true in their code during its development.
  - While this capability existed previously in Java, programmers had to add and remove the assertion code at deployment.
  - With the release of version Java 1.4 and `assert` statements, programmers can enable and disable this capability at runtime from the command line.
- An assertion is defined as "a statement containing a `Boolean` expression that the programmer believes to be true at the time the statement is executed."
  - In other words, a conditional expression returns true if the code is functioning as intended.
  - Assertions are used to test for errors that occur internally in your program and not because of user input or error.
- This chapter covers the following topics relating to assertions.
  - The syntax of assertions
  - Assertions in compiling code
  - Enabling and disabling assertions at runtime
  - When to use assertions

Assertion Syntax

- Assertion statements can be expressed in two forms.
  - The syntax for the first form follows.

    ```
    assert BooleanExpression1;
    ```

    - When the system runs the assertion, it evaluates *BooleanExpression1* and if it is false throws an `AssertionError` with no detail message.
  - The second format for assertions follows.

    ```
    assert BooleanExpression1:Expression2;
    ```

    - *BooleanExpression1* is a `Boolean` expression.
    - `Expression2` is an expression that has a value. (It cannot be an invocation of a method that is declared void.)
    - *Expression2* must be a valid argument to the `AssertionError` constructor, and therefore, be one of the following types: `Object`, `boolean`, `char`, `int`, `long`, `float`, or `double`.
    - When the system runs the assertion, it evaluates *BooleanExpression1* and if it is false throws an `AssertionError` passing the value of *Expression2* to the appropriate `AssertionError` constructor, which uses the string representation of the value as the error's detail message.
- The examples on the next page demonstrate both forms of the `assert` statement.

Assertion Syntax

AssertionA.java

```
 1. package examples.assertions;
 2.
 3. public class AssertionA {
 4.     public static void main(String[] args) {
 5.         int d = (int) Math.ceil((Math.random() * 6));
 6.         //Assert that the value is from 1 to 6
 7.         assert d >= 1 && d <= 6;
 8.         System.out.println(d);
 9.     }
10. }
```

AssertionB.java

```
 1. package examples.assertions;
 2.
 3. public class AssertionB {
 4.     public static void main(String[] args) {
 5.         int d = (int) Math.ceil((Math.random() * 6));
 6.         //Assert that the value is from 1 to 6
 7.         assert d >= 1 && d <= 6 : "Rand:" + d;
 8.         System.out.println(d);
 9.     }
10. }
```

- The `assert` statements could have also been written as:

```
if (!(assert d >= 1 && d <= 6))
    throw new AssertionError();
```

- or as:

```
if (!(assert d >= 1 && d <= 6))
    throw new AssertionError("Rand:" + d);
```

- Examples of these two variations can be found in the files named `AssertionC.java` and `AssertionD.java`, respectively.
  - A file named `AssertAsIdentifier.java` is also supplied to test code that uses assert as an identifier.

Compiling with Assertions

- The use of `assert` as a keyword was not introduced until version 1.4 of the JDK.
  - Prior to release 1.4, `assert` could have been used as an identifier, since it was not recognized as a keyword.
- The addition of the `assert` keyword to the Java programming language does not cause any problems with preexisting binaries (`.class` files).
  - However, if you try to compile an application that uses `assert` as an identifier, you will receive a warning or error message.
  - In order to ease the transition from a world where `assert` is a legal identifier to one where it is not, the compiler in JDK 1.4 introduced support for two modes of operation.
- How code is compiled (using assert, either as an identifier or as a keyword) depends on the version of Java being used.
  - Java 1.4
    - The compiler uses the older syntax (`assert` as an identifier) for backward compatibility and a warning.
    - In order to use the newer syntax (`assert` as a keyword), you must compile your code using the following command.

      `javac -source 1.4 MyClass.java`
  - Java 1.5 and higher
    - The default uses the newer syntax (`assert` as a keyword).
    - In order to use the older syntax (`assert` as an identifier), you must compile your code using the following command.

      `javac -source 1.3 MyClass.java`

Enabling and Disabling Assertions

- The choice to use an `assert` statement rather than an `AssertionError` object has the advantage because an `assert` statement can be enabled and/or disabled at runtime.
  - Assertions might be enabled during development in order to know whether certain assumptions are true.
  - At runtime, assertions can be disabled to avoid the overhead of the statements in the program yet save the developer from having to remove the statements from the final code.
- By default, assertions are disabled at runtime.
  - Several command line switches allow assertions to be selectively enabled and/or disabled.
    - The following two command line switches are equivalent and are used to enable assertions.

      ```
      -ea    -enableassertions
      ```

    - The following two command line switches are equivalent and are used to disable assertions.

      ```
      -da    -disableassertions
      ```

    - The following two command line switches are equivalent and are used to enable system assertions.

      ```
      -esa   -enablesystemassertions
      ```

    - The following two command line switches are equivalent and are used to disable system assertions.

      ```
      -dsa   -disablesystemassertions
      ```

  - The meaning and sample usage of each of the above is detailed on the next page.

Enabling and Disabling Assertions

- The following chart represents the different formats for enabling and disabling assertions from the command line.
  - The following conventions will be used in the tables below.
    - *MyApp* represents the Java program to run.
    - *MyClass* represents an individual class.
    - *my.package.name* represents a package.
  - Switch syntax examples

Switch	-ea
Example	java -ea *MyApp*
Meaning	Enables assertions in all classes except system classes

Switch	-ea:*ClassName*
Example	java -ea:*my.package.name.MyClass MyApp*
Meaning	Enables assertions in the *my.package.name.MyClass* Class only

Switch	-ea:*PackageName*
Example	java -ea:*my.package.name MyApp*
Meaning	Enables assertions in all classes in the *my.package.name* package

Switch	-ea:*PackageName*...
Example	java -ea:*my.package.name... MyApp*
Meaning	Enables assertions in all classes in the *my.package.name* package and its sub-packages

Switch	-ea:...
Example	java -ea:... *MyApp*
Meaning	Enables assertions in all classes in the unnamed package

Enabling and Disabling Assertions

- Additional switch syntax examples

Switch	`-esa`
Example	`java -esa MyApp`
Meaning	Enables assertions in system classes

Switch	`-da`
Example	`java -da MyApp`
Meaning	Disables assertions in all classes except system classes

Switch	`-da:ClassName`
Example	`java -da:my.package.name.MyClass MyApp`
Meaning	Disables assertions in the `my.package.name.MyClass` Class only

Switch	`-da:PackageName`
Example	`java -da:my.package.name MyApp`
Meaning	Disables assertions in all classes in the `my.package.name` package

Switch	`-da:PackageName...`
Example	`java -da:my.package.name... MyApp`
Meaning	Disables assertions in all classes in the `my.package.name` package and its sub-packages

Switch	`-da:...`
Example	`java -ea:... MyApp`
Meaning	Disables assertions in all classes in the unnamed package

Switch	`-dsa`
Example	`java -dsa MyApp`
Meaning	Disables assertions in system classes

- System classes are those defined in Sun's API and recognized by the JVM as "trusted" verification.

Assertion Usage

- When should a developer use an assertion instead of an exception?
  - Whereas both exceptions and assertions ensure that your code is running properly, exceptions are error checks that should never be turned off.
    - They check for conditions that are integral to the functionality of your program at deployment.
    - For example, an exception, not an assertion, would be used to check if you have the correct number of arguments on the command line.
  - Assertions are typically used at development to ensure that what a programmer believes to be happening actually is happening.
    - For example, checking that a connection to a resource exists before writing to it or verifying that a reference is not equal to null before using it to invoke a method
- The following chart provides some guidelines to help you decide when to use assertions versus exceptions.

Assertions	Exceptions
To enforce internal assumptions about data structures	To enforce correct usage on the command line
To enforce constraints on arguments in private methods	To enforce constraint on arguments to public, protected or package methods
To check conditions at the beginning of a method invocation	To enforce usage of the public interface or particular protocols
To check for conditional cases that should never happen especially if the programmer assumes they can not happen	To check for conditional cases that are expected to happen
To check for conditions at the end of a method invocation	To enforce the integrity of user input

Exercises

1. Determine which command line switches are necessary and compile the following programs used in this chapter.

   - `AssertionA.java`
   - `AssertionB.java`
   - `AssertionC.java`
   - `AssertionD.java`
   - `AssertAsIdentifier.java`

2. Place a check mark next to each of the applications that can be compiled using the Java compiler from JDK 1.3.

   ___ `AssertionA.java`
   ___ `AssertionB.java`
   ___ `AssertionC.java`
   ___ `AssertionD.java`
   ___ `AssertAsIdentifier.java`

3. Place a check mark next to each of the applications that can be compiled using the Java compiler from JDK 1.4 or higher.

   ___ `AssertionA.java`
   ___ `AssertionB.java`
   ___ `AssertionC.java`
   ___ `AssertionD.java`
   ___ `AssertAsIdentifier.java`

4. Place a check mark next to each of the applications that can be run using Java 1.4 or higher.

   ___ `AssertionA.java`
   ___ `AssertionB.java`
   ___ `AssertionC.java`
   ___ `AssertionD.java`
   ___ `AssertAsIdentifier.java`

Chapter 5:
Regular Expressions

1) Regular Expressions 5-2

2) String Literals 5-4

3) Character Classes 5-5

4) Quantifiers 5-9

5) Capturing Groups and Backreferences 5-11

6) Boundary Matchers 5-12

7) Pattern and Matcher 5-13

Regular Expressions

- Regular expressions are a way to describe a set of strings based on common characteristics shared by each string in the set.
  - They can be used to search, edit, or manipulate text and data.
  - The specific syntax to create regular expressions is different from the normal syntax of the Java programming language.
  - Regular expressions vary in complexity, but once you understand the basics of how they are constructed, you will be able to decipher (or create) any regular expression.
- There are many different flavors of regular expressions such as grep, Perl, Tcl, Python, PHP, and awk.
  - The regular expression syntax in the `java.util.regex` API (introduced in Java SE 1.4) is most similar to that of Perl.
- The `java.util.regex` package primarily consists of the following three classes.
  - A `Pattern` object is a compiled representation of a regular expression.
  - A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string.
  - A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.
- Before discussing the above classes in detail, we will create a program to test the syntax of a regular expression, using the `matches` method of the `String` class.

Regular Expressions

RegExTester1.java

```
 1. package examples.regex;
 2. import java.io.*;
 3.
 4. public class RegExTester1 {
 5.
 6.     private static BufferedReader kb =
 7.         new BufferedReader(
 8.             new InputStreamReader(System.in));
 9.
10.     public static void main(String[] args) {
11.         String txt = null;
12.         String regex = null;
13.         boolean found = false;
14.         String response = null;
15.         while (true) {
16.             regex = prompt("Enter your regex: ");
17.             txt = prompt("Enter String to match: ");
18.             found = txt.matches(regex);
19.             response = "Match Found\n";
20.             if(!found)
21.                 response ="No " + response;
22.             print(response);
23.         }
24.     }
25.
26.     private static String prompt(String s){
27.         String userText = null;
28.         try{
29.             print(s);
30.             userText = kb.readLine();
31.         }catch (IOException ioe){
32.             ioe.printStackTrace();
33.         }
34.         return userText;
35.     }
36.
37.     private static void print(String s){
38.         System.out.print(s);
39.     }
40. }
```

String Literals

- The simplest form of pattern matching supported is the match of a string literal.
  - For example, testing the regular expression of "`hello`" and an input string of "`hello`" with the `RegExTester` application will succeed because the strings are identical.
- The `java.util.regex` API also supports a number of special characters that affect the way a pattern is matched.
- These special characters are called metacharacters.
  - Metacharacters are characters with a special meaning interpreted differently by the matcher than regular characters.
  - The metacharacters supported are:

```
(  [  {  \  ^  -  $  |  ]  }  )  ?  *  +  .
```

- Change the regular expression to "`.ello`" and the input string to "`hello.`"
  - The match will still succeed because the "`.`" metacharacter is interpreted as "any character."
  - For this reason, an input string of "`jello`" would also match.
  - Preceding a metacharacter with a backslash will force the metacharacter to be treated as an ordinary character, if desired.
- In certain situations, the special characters listed above will not be treated as metacharacters.
  - This will become evident as more is learned about how regular expressions are constructed.

Character Classes

- A character class is a set of characters enclosed within square brackets.
  - It specifies the characters that will successfully match a single character from a given input string.
- Simple class
  - The simplest form of a character class is a set of characters side-by-side within square brackets.
    - For example, the regular expression "`[dmr]ice`" will match the words "`dice`", "`mice`", or "`rice`", but not "`vice`", because it defines a character class (accepting either "`d`", "`m`", or "`r`") as its first character.
- Negation
  - To match all characters except those listed, insert the "`^`" metacharacter at the beginning of the character class.
    - For example, the regular expression "`[^dmr]ice`" will not match the words "`dice`", "`mice`", or "`rice`", but will match the words "`nice`" or "`vice`" because it defines a character class (rejecting either "`d`", "`m`", or "`r`") as its first character.
- Ranges
  - A character class representing a range of values, such as the letters "a through m" can be specified by inserting the "`-`" metacharacter between the first and last character to be matched, such as "`[a-m]`."

Character Classes

- Ranges - continued
  - You can also place different ranges beside each other within the class to expand the match possibilities such as the range "`[a-mA-M]`."
    - For example, the regular expression "`[a-mA-M]ice`" will match the words "`dice`", "`Dice`", "`mice`", or "`Mice`", but will not match the words "`nice`" or "`Vice`".
- Unions
  - A union is a single character class comprised of two or more separate character classes.
  - To create a union, simply nest one character class inside another, such as "`[a-m[A-M]]`". This particular union would be equivalent to the range "`[a-mA-M]`."
- Intersections
  - To create a single character class matching only the characters common to all of its nested classes, use "`&&`", as in "`[a-p&&[m-z]]`".
  - This particular intersection creates a single character class matching only the letters common to both character classes: m,n,o, and p.
    - For example, the regular expression "`[a-p&&[m-z]]ice`" will match the words "`mice`", "`nice`", but not "`dice`", "`rice`" or "`vice`."
    - This is because neither "`d`", "`r`", nor "`v`" is common to both the character classes "`[a-p]`" and "`[m-z]`."

Character Classes

- Subtraction
  - Subtraction can be used to negate one or more nested character classes, such as "`[a-z&&[^m-r]]`."
    - For example, the regular expression "`[a-z&&[^m-r]]ice`" will match "`dice`" or "`vice`" but not "`mice`" or "`rice`."
- It is important to note that certain metacharacters, which are in effect inside a character class, might be treated as regular characters outside a character class and vice versa.
  - The regular expression "`.`" loses its special meaning inside a character class.
  - The regular expression "`-`" becomes a range forming metacharacter inside a character class.
- The `java.util.regex` API provides a number of character classes, which offer convenient shorthands for commonly used regular expressions.
  - These character classes are summarized in a table on the following page.

Character Classes

Predefined character classes	
Construct	Matches:
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

POSIX character classes (US-ASCII only)	
Construct	Matches:
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character: [A-Z]
\p{ASCII}	All ASCII: [\x00-\x7F]
\p{Alpha}	An alphabetic character: [\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character: [\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}\x20]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]

Classes for Unicode blocks and categories	
Construct	Matches:
\p{InGreek}	A character in the Greek block (simple block)
\p{Lu}	An uppercase letter (simple category)
\p{Sc}	A currency symbol
\P{InGreek}	Any character except one in the Greek block (negation)

Quantifiers

- Quantifiers allow you to specify the number of occurrences against which to match.

- There are subtle differences among the three defined categories of quantifiers as described below.
  - Greedy quantifiers
    - They are considered "greedy" because they force the matcher to read in the entire input string prior to attempting the first match.
    - If the first match attempt (the entire input string) fails, the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left from which to back off.
    - Depending on the quantifier used in the expression, the last thing it will try matching against is 1 or 0 characters.
  - Reluctant quantifiers
    - These take the opposite approach of greedy quantifiers.
    - They start at the beginning of the input string, and then reluctantly consume one character at a time looking for a match.
    - The last thing they try is the entire input string.
  - Possessive quantifiers
    - They always consume the entire input string, trying once (and only once) for a match.
    - Unlike the greedy quantifiers, possessive quantifiers never back off.

Quantifiers

- The table below summarizes the syntax and meaning of the various quantifiers that can be used in regular expressions.

Quantifiers			Meaning
Greedy	Reluctant	Possessive	
`X?`	`X??`	`X?+`	`X`, once or not at all
`X*`	`X*?`	`X*+`	`X`, zero or more times
`X+`	`X+?`	`X++`	`X`, one or more times
`X{n}`	`X{n}?`	`X{n}+`	`X`, exactly `n` times
`X{n,}`	`X{n,}?`	`X{n,}+`	`X`, at least `n` times
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	`X`, at least `n` but not more than `m` times

- An example of a regular expression using one of the greedy quantifiers from the table above is shown below.
  - The regular expression "`[abc]{3}`" will match any of the following 27 strings.

```
"aaa"    "aab"    "aac"
"aba"    "abb"    "abc"
"aca"    "acb"    "acc"
"baa"    "bab"    "bac"
"bba"    "bbb"    "bbc"
"bca"    "bcb"    "bcc"
"caa"    "cab"    "cac"
"cba"    "cbb"    "cbc"
"cca"    "ccb"    "ccc"
```

Capturing Groups and Backreferences

- Capturing groups can be used to treat multiple characters as a single unit.
  - They are created by placing the characters to be grouped inside a set of parentheses.
  - For example, the regular expression "`(cat)`" creates a single group containing the letters "`c`" "`a`" and "`t`."
  - The portion of the input string that matches the capturing group will be saved in memory for later recall using a backreference.
- A backreference is specified in the regular expression as a backslash "`\`" followed by a digit, indicating the number of the group to be recalled.
- The regular expression below uses a capturing group and backreference within the same definition.
  - The regular expression of "`(hello)goodbye\1`" would match the string of "`hellogoodbyehello`."

Boundary Matchers

- You can make your pattern matches more precise by specifying such information with boundary matchers. For example:
  - finding a particular word, at the beginning or end of a line; or
  - matching on a word boundary, or at the end of the previous match.
- The following table lists each of the boundary matchers.

Boundary Matchers	
Construct	Matches:
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Pattern and Matcher

- As mentioned at the beginning of this chapter, the `java.util.regex` package primarily consists of the following three classes.
  - A `Pattern` object is a compiled representation of a regular expression.
  - A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string.
  - A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.
- Regular expression support also exists in the `String` class through several methods that mimic the behavior of `java.util.regex.Pattern`.
  - Some of the methods from the `String` class are listed below.
    - `public boolean matches(String regex)`
    - `public String[] split(String regex, int max)`
    - `public String[] split(String regex)`
    - `public String replaceAll(String regex, String replacement)`
- The `Pattern` is a factory class that provides a static method named `compile` to obtain a `Pattern` object.
  - From the `Pattern` object, the static `matcher` method is then called to obtain a `Matcher` object.
  - The example on the next page uses a typical invocation of the above sequence of steps.

Pattern and Matcher

- The example below is an excerpt from `RegExTester2.java`. It expands, in the following ways, upon the `RegExTester1.java`, which was shown earlier.
  - It uses the `Pattern` and `Matcher` classes to construct and test a given regular expression.
  - It uses the `find` method of the `Matcher` class to find each occurrence of the regular expression within the given string.
  - It uses the `group`, `start`, and `end` methods of the `Matcher` class to obtain more information about each match.

RegExTester2.java

```
 1. public static void main(String[] args) {
 2.     Pattern p = null;
 3.     Matcher m = null;
 4.     boolean found = false;
 5.     String prompt1 = "Enter your regex: ";
 6.     String prompt2 = "Enter String to search: ";
 7.     while (true) {
 8.         p = Pattern.compile(prompt(prompt1));
 9.         m = p.matcher(prompt(prompt2));
10.         found = false;
11.         while (m.find()) {
12.             print("I found the text ");
13.             print(m.group());
14.             print(" starting at index ");
15.             print("" + m.start());
16.             print(" and ending at index ");
17.             print("" + m.end() + "\n");
18.             found = true;
19.         }
20.         if(!found){
21.             print("No match found.\n");
22.         }
23.     }
24. }
```

Exercises

1. Create a `Pattern` object that can test the format of a date.
   - The date should be of the form *mm/dd/yyyy*.
   - The month and day may be one or two digits.
2. Create another `Pattern` similar to the one above that allows the delimeter to be any non-word character rather than just a "/".
3. Write an application that lists all of the files in a directory that match a given regular expression.
   - Both the name of the directory and regular expression to match should be given as command line arguments to the application.
4. Create another version of the preceding application that recursively searches all sub-directories of the given directory.
5. Modify the `EnumeratedAddress` class created earlier so that the methods/constructors permitting a zip code to be passed only accept zip codes of the form `#####` or `#####-####` where # represents a digit.
   - If the zip code does not match the pattern, the code should throw a `java.util.UnknownFormatConversionException`.

This Page Intentionally Left Blank

Chapter 6:
The Java Collection Classes

1) Introduction 6-2

2) The Arrays Class 6-3

3) Searching and Sorting Arrays of Primitives 6-4

4) Sorting Arrays of Objects 6-5

5) The `Comparable` and `Comparator` Interfaces 6-6

6) Sorting - Using `Comparable` 6-7

7) Sorting - Using `Comparator` 6-10

8) Collections 6-11

9) Lists and Sets 6-12

10) Iterators 6-13

11) Lists and Iterators Example 6-14

12) Maps 6-17

13) Maps and Iterators Example 6-18

14) The Collections Class 6-20

15) Rules of Thumb 6-23

Introduction

- The `java.util` package contains a set of classes referred to as the **Collections Framework**.
  - In this chapter, we will explore the various interfaces, abstract classes, and concrete classes in this framework.
- The Java Collections Framework has much of the same functionality as the C++ Standard Template Library.
- You may already be familiar with some collection classes such as `Vector` and `Hashtable`.
  - These older classes may still be used, but the new classes have significant advantages including:
    - reduced programming effort;
    - support for software reuse; and
    - increased program speed.
- All of the collection classes are heterogeneous (i.e., they may contain any kinds of objects), but they may contain objects only and not primitive types.
  - With the introduction of Java SE 5, **generics** provided a means of specifying the types of objects a collection may contain.
    - This will be discussed in detail in the chapter on generics.
- Collection classes are not thread-safe.
  - Older classes, such as `Hashtable` and `Vector`, were thread-safe, but there was no way to turn off that mechanism.
    - Thread safety in regards to collections will be discussed in a later chapter pertaining to threads.

The Arrays Class

- Since the Collection Framework classes can only hold objects, arrays are still essential data structures for primitive types.

- The `java.util.Arrays` class includes several useful `static` methods for processing arrays.
  - `binarySearch()`
    - Searches a sorted array for a specified value
    - Returns the zero-based index of the search value if found; otherwise returns (`-pos - 1`), where `pos` is the index of the position it should occupy
  - `equals()`
    - Returns `true` if two arrays contain the same elements in the same order
    - Also returns `true` if both array references are `null`
  - `fill()`
    - Assigns a specified value to each element of an array (or to each element in a specified range of the array)
  - `sort()`
    - Sorts an array or a specified range of an array
    - When used with arrays of primitives, the sorting algorithm is an adaptation of **quicksort**.
    - It sorts in ascending order only. If you want some other ordering scheme, you have to use arrays of objects.

Searching and Sorting Arrays of Primitives

- This program demonstrates methods in the `Arrays` class.

ArraysTest.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class ArraysTest {
 5.     public static void main(String[] args) {
 6.         int[] a = { 88, 17, -10, 34, 27, 0, -2 };
 7.         int[] find = { -10, -1, 0, 1, 34, 100 };
 8.         System.out.print("Before sort: ");
 9.         System.out.println(Arrays.toString(a));
10.
11.         Arrays.sort(a);
12.         System.out.print("After sort: ");
13.         System.out.println(Arrays.toString(a));
14.
15.         System.out.print("Looking For: ");
16.         System.out.println(Arrays.toString(find));
17.         for (int val : find)
18.             search(a, val);
19.     }
20.
21.     private static void search(int[] arr, int n) {
22.         int pos = Arrays.binarySearch(arr, n);
23.         String msg;
24.         if (pos >= 0) {
25.             msg = "Found " + n + ", position " + pos;
26.         } else {
27.             int insertionPoint = -(pos + 1);
28.             if (insertionPoint == arr.length) {
29.                 msg = "Append " + n + " to end";
30.             } else {
31.                 msg = "Insert " + n + " at position "
32.                         + insertionPoint;
33.             }
34.         }
35.         System.out.println(msg);
36.     }
37. }
```

Sorting Arrays of Objects

- A **mergesort** algorithm is used when sorting an array of objects.

- There are two ways to specify an ordering scheme when sorting an array of objects.
  - The first way of sorting an array of objects relies upon the **natural ordering** of the objects in the array.
    - The objects in the array must implement the `Comparable` interface, which is used to define the natural ordering.
    - The two `sort` methods that rely on the natural ordering are shown below.

```
sort(Object[] a)
sort(Object[] a, int fromIndex, int toIndex)
```

  - The second way of sorting an array of objects relies upon an additional parameter being passed to the sort method.
    - The additional parameter must be an object that implements the `Comparator` interface, and as such defines the ordering scheme.
    - The two sort methods that rely upon a `Comparator` object are shown below.

```
sort(Object[] a, Comparator c)
sort(Object[] a, int fromIndex,
     int toIndex, Comparator c)
```

- The `Comparable` and `Comparator` interfaces are explained in more detail on the following pages.

The `Comparable` and `Comparator` Interfaces

- The `Comparable` interface declares a single method as shown below.

```
public interface java.lang.Comparable {
   int compareTo (Object o);
}
```

  - The method is required to return:
    - a negative integer if the object the method is called on precedes `o` in ordering;
    - `0` (zero) if the object the method is called on is equal to o; or
    - a positive integer.
  - The `String`, `Date`, and all the primitive wrapper classes are just some of the classed that already implement `Comparable`.

- The `Comparator` interface declares a single method as shown below.

```
public interface java.util.Comparator {
   int compare (Object o1, Object o2);
}
```

  - The method is required to return:
    - a negative number if `o1` precedes `o2` in the ordering;
    - `0` (zero) if they are equal; or
    - a positive number otherwise.

Sorting - Using `Comparable`

- The `Fraction` class below defines its natural ordering by implementing the `Comparable` interface.

Fraction.java

```
 1. package examples.collections;
 2.
 3. public class Fraction implements Comparable {
 4.     private int n, d;
 5.
 6.     public Fraction(int n, int d) {
 7.         this.n = n;
 8.         this.d = d;
 9.     }
10.
11.     public String toString() {
12.         return n + "/" + d;
13.     }
14.
15.     public int compareTo(Object o) {
16.         //System.out.print("Comparing: ");
17.         //System.out.println (this + " and " + o);
18.         Fraction f1 = (Fraction) o;
19.         double hostVal = (double) n / d;
20.         double paramVal = (double) f1.n / f1.d;
21.         if (hostVal == paramVal)
22.             return 0;
23.         else if (hostVal < paramVal)
24.             return -1;
25.         else
26.             return 1;
27.     }
28. }
```

  - The `compareTo` method defined above will be called repetitively by the `Arrays.sort()` method when an array of `Fraction` objects is sorted.

- The application on the next page can be run to test the above `Fraction` class.

Sorting - Using `Comparable`

FractionTest1.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class FractionTest1 {
 5.     public static void main(String a[]) {
 6.         Fraction[] f = { new Fraction(-2, 3),
 7.            new Fraction(1, 10), new Fraction(-4, 5),
 8.            new Fraction(1, 50), new Fraction(2, 3)};
 9.         Arrays.sort(f);
10.         System.out.println(Arrays.toString(f));
11.     }
12. }
```

- The output generated by running the above application is shown below.

```
[-4/5, -2/3, 1/50, 1/10, 2/3]
```

- Un-commenting the print statements in the `compareTo` method of the `Fraction` class produces the following output.

```
Comparing: -2/3 and 1/10
Comparing: 1/10 and -4/5
Comparing: -2/3 and -4/5
Comparing: 1/10 and 1/50
Comparing: -2/3 and 1/50
Comparing: 1/10 and 2/3
[-4/5, -2/3, 1/50, 1/10, 2/3]
```

  - The existence of the additional output indicates that some method is calling the `compareTo` method of the `Fraction` class.
  - The `sort` method of the `Arrays` class is the method that calls `compareTo` as needed by its sorting algorithm.

Sorting - Using `Comparator`

- The class below implements `Comparator`, whose compare method is capable of sorting in either direction.
  - The direction will be passed to the class' constructor.
  - The value passed as the parameter uses an enum type named `SortOrderEnum`.

FractionComparator.java

```
 1. package examples.collections;
 2.
 3. import java.util.Comparator;
 4.
 5. public class FractionComparator implements Comparator
 6. {
 7.     private SortOrderEnum order;
 8.     public FractionComparator(SortOrderEnum so){
 9.         order = so;
10.     }
11.     public int compare(Object obj1, Object obj2) {
12.         Fraction f1 = (Fraction) obj1;
13.         Fraction f2 = (Fraction) obj2;
14.         if(order == SortOrderEnum.LOW_TO_HIGH){
15.             return f1.compareTo(f2);
16.         }else{
17.             return f2.compareTo(f1);
18.         }
19.     }
20. }
```

SortOrderEnum.java

```
1. package examples.collections;
2. public enum SortOrderEnum {
3.     LOW_TO_HIGH, HIGH_TO_LOW;
4. }
```

- The application on the following page is an updated version of `FractionTest1` that includes use of the classes above.

Sorting - Using `Comparator`

FractionTest2.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class FractionTest2 {
 5.     public static void main(String a[]) {
 6.         Fraction[] f = { new Fraction(-2, 3),
 7.            new Fraction(1, 10), new Fraction(-4, 5),
 8.            new Fraction(1, 50), new Fraction(2, 3)};
 9.
10.         //Same as previous code
11.         //  relies on natural ordering and only
12.         //  works because Fraction implements the
13.         //  Comparable interface
14.         Arrays.sort(f);
15.         System.out.println(Arrays.toString(f));
16.
17.         // Now sort the array from high to low
18.         //   requires second parameter to be an
19.         //   object that implements Comparator
20.         SortOrderEnum hl = SortOrderEnum.HIGH_TO_LOW;
21.         Arrays.sort(f,new FractionComparator(hl));
22.         System.out.println(Arrays.toString(f));
23.
24.         // Now sort the array from low to high
25.         //   also using a comparator
26.         SortOrderEnum lh = SortOrderEnum.LOW_TO_HIGH;
27.         Arrays.sort(f, new FractionComparator(lh));
28.         System.out.println(Arrays.toString(f));
29.     }
30. }
```

Collections

- The collections framework provides two families of containers: a `Collection` and a `Map`.

- A `Collection` is an unordered group of elements with operations for insertion, removal, and testing of membership.
  - The operations are specified by the `Collection` interface, which includes the following.
    - `add()` – add a new element
    - `remove()` – remove an element
    - `size()` – return number of elements
    - `isEmpty()` – determine if collection is empty
    - `iterator()` – get an `Iterator` object
    - `contains()` – determine if collection contains an element
    - `toArray()` – return all the elements as an array

- The `Collection` interface is further refined into the `Set` and `List` interfaces.
  - The `Set` interface is identical to the `Collection` interface, with the added specification that duplicates are not allowed.
  - The `List` interface defines an ordered sequence that supports random access and bi-directional traversal. Methods in the `List` interface include the following.
    - `get()` – get element at specified index
    - `indexOf()` – get index of specified element
    - `listIterator()` – get a `ListIterator` object

Lists and Sets

- Two of the implementations of `List` provided by Java are `ArrayList` and `LinkedList`.
  - `ArrayList` is an expandable array and performs better when you need random access to the list elements.
    - `ArrayList` extends `AbstractList`, an abstract class that implements some (but not all) of the methods in the `List` interface.
    - `AbstractList` is an example of a "**convenience**" class, because it is designed to make it easy for you to write your own implementation of a "random-access" list. To do so, simply extend the class and write the remaining methods.
    - `Vector` also implements the `List` interface, but `ArrayList` is preferred for new code.
  - `LinkedList` implements a doubly-linked list and is optimized for inserting and removing elements anywhere in a list.
    - `LinkedList` extends `AbstractSequentialList`, another "convenience" class.
- Two of the implementations of `Set` provided by Java are `HashSet` and `TreeSet`.
  - `HashSet` implements `Set` and as such, does not permit duplicates.
    - It does not guarantee that the order of its elements will remain constant over time.
  - `TreeSet` implements `Set` and as such, does not permit duplicates.
    - A `TreeSet` maintains its elements in sorted order according the natural ordering of its elements.

Iterators

- The `Iterator` and `Enumeration` interfaces are used to facilitate traversing through a `Collection`.
  - The functionality of `Enumerator` interface is duplicated by the `Iterator` interface.
  - In addition, `Iterator` adds an optional remove operation and has shorter method names.
  - New implementations should consider using `Iterator` in preference to `Enumeration`.
  - The Iterator interface is shown below.

```
public interface java.util.Iterator {
   boolean hasNext();
   Object  next();
   void    remove();
}
```

- The `List` interface supports a `ListIterator`.
  - A `ListIterator` extends an `Iterator` by allowing bi-directional traversal and the ability to add to a list.
  - Some of the additional methods in the `ListIterator` interface include the following.

```
boolean hasPrevious();
Object  previous();
void    add (Object o);
```

- Note that if a `Collection` changes during the lifetime of an `Iterator`, other than by operations performed through the `Iterator` itself, the `Iterator` becomes invalid, and any subsequent use of the `Iterator` will result in a `ConcurrentModificationException`.

Lists and Iterators Example

- The example below defines several static methods that print information about a `List` passed as a parameter.

ListUtils.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class ListUtils {
 5.
 6.     // Use an Iterator to print all elements in List
 7.     public static void outputList(List list){
 8.         Iterator i = list.iterator();
 9.         while (i.hasNext()) {
10.             print(i.next());
11.         }
12.     }
13.
14.     // Find first occurence of an object in List
15.     public static void search(List list, Object obj){
16.         int index = list.indexOf(obj);
17.         if (index != -1) {
18.             print(obj + " found in pos:" + index);
19.         } else {
20.             print("Didn't find " + obj);
21.         }
22.     }
23.
24.     // Helper method for printing an object
25.     private static void print(Object obj){
26.         System.out.println(obj);
27.     }
28. }
```

  - Note that the methods above are defined to take a `List` as a parameter.
    - This permits code that utilizes these methods to pass an `ArrayList`, a `LinkedList`, or any other implementation of `List`.
    - This is demonstrated on the following page.

Lists and Iterators Example

ListTest.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class ListTest {
 5.     public static void main(String[] args) {
 6.         // Build ArrayList with an initial capacity
 7.         ArrayList arrayList = new ArrayList(10);
 8.         arrayList.add(new String("Java"));
 9.         arrayList.add(new String("Perl"));
10.         arrayList.add(new String("C++"));
11.
12.         // Retrieve by index
13.         print(arrayList.get(2) + " found in pos:2");
14.
15.         // Search for an element in List
16.         print("\nSearch Results:");
17.         ListUtils.search(arrayList, "Perl");
18.
19.         // Traverse using an iterator
20.         print("\nIterator Results...");
21.         ListUtils.outputList(arrayList);
22.
23.         // Build LinkedList from ArrayList
24.         LinkedList list = new LinkedList(arrayList);
25.
26.         // Manipulate contents of the LinkedList
27.         list.remove("Perl");
28.         list.add("JavaScript");
29.
30.         // Traverse using an iterator
31.         print("\nIterator Results...");
32.         ListUtils.outputList(list);
33.
34.         // Traverse in reverse order
35.         ListIterator listIter =
36.             list.listIterator(list.size());
37.         print("\nIterating in reverse...");
38.         while (listIter.hasPrevious()) {
39.             print(listIter.previous());
40.         }
```

▸ Continued on following page

List and Iterators Example

ListTest.java *continued*

```
41.
42.          // Traverse using enhanced for loop
43.          print("\nFor Loop Results...");
44.          for(Object obj:list)
45.              print(obj);
46.      }
47.
48.      // Helper method for printing an object
49.      private static void print(Object obj){
50.          System.out.println(obj);
51.      }
52.  }
```

- The example below demonstrates the potential problem(s) when dealing with more than one `Iterator` obtained from the same object.

ConcurrentModificationProblem.java

```
 1.  package examples.collections;
 2.  import java.util.*;
 3.
 4.  public class ConcurrentModificationProblem {
 5.      public static void main(String[] args) {
 6.          // Build ArrayList with an initial capacity
 7.          ArrayList arrayList = new ArrayList(10);
 8.          arrayList.add(new String("Java"));
 9.          arrayList.add(new String("Perl"));
10.          arrayList.add(new String("C++"));
11.
12.          Iterator iterA = arrayList.iterator();
13.          Iterator iterB = arrayList.iterator();
14.
15.          iterA.next();
16.          iterA.remove();
17.
18.          // The following line will result in a
19.          // ConcurrentModificationException at runtime
20.          iterB.next();
21.      }
22.  }
```

Maps

- The `Map` interface defines a means of mapping keys to values.
  - A `Map` provides fast search time for keys.
  - Other languages often described this relationship of keys and values as a dictionary or an associative array.
  - A `Map` cannot contain duplicate keys.
  - The `Map` interface provides three collection views that allow a map's contents to be viewed on a:
    - set of keys;
    - set of key-value mappings; and
    - collection of values.
  - Some of the methods in the `Map` interface are listed below.
    - `public Object put(Object key, Object value);`
    - `public Object get(Object key);`
    - `public Set keySet();`
    - `public Set entrySet();`
    - `public Collection values();`
- Two implementations provided by Java for a `Map` are:
  - `HashMap`
    - This class makes no guarantees as to the order of the map. In particular, it does not guarantee that the order will remain constant over time.
  - `TreeMap`
    - This class guarantees that the map will be in ascending key order, according to the natural ordering of the keys.

Maps and Iterators Example

- The example below populates a `HashMap` from a two dimensional array of strings and iterates through it using a variety of techniques.
  - The data being stored are states (key) and their capital (value).

StateCaps.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class StateCaps {
 5.     final static String STATE_CAPS [][] =
 6.         {{"Alabama", "Montgomery"},
 7.            /* complete array not shown */
 8.          {"Hawaii", "Honolulu"}, {"Idaho", "Boise"}};
 9.
10.     public static void main(String args[]) {
11.         Map m = new HashMap();
12.
13.         // Populate HashMap from array
14.         for(String sc[]:STATE_CAPS){
15.             m.put(sc[0], sc[1]);
16.         }
17.
18.         // Iterate through keys
19.         System.out.println("\n----Keys------------");
20.         Iterator iter = m.keySet().iterator();
21.         while (iter.hasNext()){
22.             System.out.println(iter.next());
23.         }
24.
25.         // Loop through values
26.         System.out.println("\n----Values----------");
27.         for(Object obj:m.values()){
28.             System.out.println(obj);
29.         }
```

continued on following page

Maps and Iterators Example

StateCaps.java - continued

```
30.
31.         // Iterate through entries
32.         System.out.println("\n----Entries---------");
33.         iter = m.entrySet().iterator();
34.         Map.Entry e;
35.         while (iter.hasNext()){
36.             e = (Map.Entry)iter.next();
37.             System.out.print(e.getKey() + ", ");
38.             System.out.println(e.getValue());
39.         }
40.
41.         // Return capitals of states supplied as
42.         // command line arguments
43.         System.out.println("\n--------------------");
44.         for (String s:args){
45.             System.out.print("Capital of " + s);
46.             System.out.println(" is " + m.get(s));
47.         }
48.
49.         // Convert to TreeMap where the keys will
50.         // then be maintained in sorted order.
51.         System.out.println("\n----Sorted Entries--");
52.         TreeMap tm = new TreeMap(m);
53.         iter = tm.entrySet().iterator();
54.         while (iter.hasNext()){
55.             e = (Map.Entry)iter.next();
56.             System.out.print(e.getKey() + ", ");
57.             System.out.println(e.getValue());
58.         }
59.     }
60. }
```

The Collections Class

- Like the `Arrays` class, the `java.util.Collections` class has static methods for performing common tasks on a `Collection`.
  - Many of the methods in the `Collections` class (such as `sort`) take a `List` as a parameter rather than a `Collection`, because of the assumption of ordering.
  - Other methods that take a `Collection` as a parameter (such as `min` and `max`), are applicable to all types of collections.
- The example below is written to:
  - create an `ArrayList` of 100 random numbers from 0 to 49 inclusive;
  - determine the highest number generated; and
  - determine the number of times the highest number was generated.

HighestRandom.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class HighestRandom {
 5.
 6.     public static void main(String[] args) {
 7.         ArrayList a = new ArrayList(100);
 8.         Random r = new Random();
 9.         for(int i = 0; i < 100; i++){
10.             a.add(r.nextInt(50));
11.         }
12.         Object obj = Collections.max(a);
13.         System.out.println("largest: " + obj);
14.         int freq = Collections.frequency(a, obj);
15.         System.out.println("frequency: " + freq);
16.     }
17. }
```

The Collections Class

- The example on the following page expands upon the previous example in the following ways.
  - Instead of only calculating the frequency for the highest value, it calculates the frequency for all of the values.
  - In order to keep track of the frequencies calculated, the results will be stored in a `TreeMap`, where the generated number will act as the key and its frequency as the value.
  - Since the keys being used are `Integer` objects, the natural ordering of the keys in a default `TreeMap` would be maintained lowest to highest.
    - The code demonstrates one technique of reversing the natural ordering of the keys through the `reverseOrder` method in the `Collections` class, which returns a `Comparator`.
    - The `TreeMap` will then be instantiated by passing a reference to the `Comparator` to its constructor.

The Collections Class

RandomFrequencies.java

```
 1. package examples.collections;
 2. import java.util.*;
 3.
 4. public class RandomFrequencies {
 5.
 6.     public static void main(String[] args) {
 7.         ArrayList a = new ArrayList();
 8.         Random r = new Random();
 9.         for(int i = 0; i < 100; i++){
10.             a.add(r.nextInt(50));
11.         }
12.
13.         // Copy ArrayList of random numbers into
14.         // a HashSet in order to remove duplicates,
15.         // leaving only the numbers that actually got
16.         // generated
17.         HashSet uniqueSet = new HashSet(a);
18.
19.         // Create a Treemap whose keys will be
20.         // sorted from high to low
21.         // (opposite of normal)
22.         TreeMap tm =
23.             new TreeMap(Collections.reverseOrder());
24.
25.         // Calculate the frequency of each number
26.         // and map it as value to number
27.         for(Object num:uniqueSet){
28.             int frq = Collections.frequency(a, num);
29.             tm.put(num, frq);
30.         }
31.
32.         // Display contents of TreeMap
33.         System.out.println("Number\tFrequency");
34.         for(Object obj:tm.entrySet()){
35.             Map.Entry me = (Map.Entry) obj;
36.             System.out.print(me.getKey());
37.             System.out.print("\t");
38.             System.out.println(me.getValue());
39.         }
40.     }
41. }
```

Rules of Thumb

- Program in terms of interface types rather than implementation types.
  - For example, prefer the former to the latter.

```
List list = new ArrayList();
ArrayList list = new ArrayList();
```

  - This is recommended in order to lessen dependence on a particular implementation. For instance:
    - `ArrayList` might later change to `LinkedList`; or
    - a program might come to depend on added methods found only in `ArrayList` and not in the `List` interface.
- When passing `Collection` types as parameters to some called method, use the least specific type.
  - For example, use `Collection` instead of `List`, or `List` instead of `ArrayList`.
  - This promotes generality and guards against depending on added methods.
- For method return types, use specific implementation types.
  - For example, `ArrayList` has very different performance metrics than `LinkedList`.
  - Because of this, the user of a method needs to know just what type of data structure is being returned in order to write more efficient code.
  - The returned type can always be converted to a more general type by the user, if desired.

Exercises

1. For this exercise, two starter files are supplied to you in the starters directory for this chapter.
   - The two classes that are supplied are named `Product.java` and `TestProduct.java`.
   - The `Product` class as currently written is only in skeletal form and requires that you provide the actual code inside each of the methods of the class.
   - The `TestProduct` class is complete and ready to be used to test your `Product` class.
2. Building upon the previous exercise, the following starter files are supplied for this exercise.
   - `Database.java`
     - This interface complete and ready to be used.
   - `ProductDBFromFile.java`
     - This class is in skeletal form and requires you to provide the concrete implementations of the declared methods.
   - `TestProductDBFromFile.java`
     - This application is complete and can be used to test your `ProductDBFromFile` class.
   - Additional information can be found in the comments of the source code for each of the above classes.
3. Update the `Product` class so that it is `Comparable`.
   - The natural ordering of a `Product` will be its `id` for the purpose of this exercise.
   - Modify the `TestProductDBFromFile` application so that it prints out the `Product` objects in sorted order.

Chapter 7:
Generics

1) Introduction 7-2

2) Defining Simple Generics 7-4

3) Generics and Subtyping 7-5

4) Wildcards 7-6

5) Bounded Wildcards 7-8

6) Generic Methods 7-9

Introduction

- When you take an element out of a `Collection`, you must cast it to the type of element that is stored in the collection.
  - This tends to be both inconvenient and unsafe.
  - The compiler does not check that your cast is the same as the collection's type, so the cast can fail at run time.
- The code below iterates through a `Collection` of `String` objects to determine the total length of the strings in the collection without the use of generics.

WithoutGenerics.java

```
 1. package examples.generics;
 2. import java.util.*;
 3.
 4. public class WithoutGenerics {
 5.     private static int length(Collection c) {
 6.         Iterator i = c.iterator();
 7.         int sum = 0;
 8.         while(i.hasNext()){
 9.             sum += ((String) i.next()).length();
10.         }
11.         return sum;
12.     }
13. }
```

- Generics (introduced in Java SE 5) provides a way for you to specify the type of a collection to the compiler, so that it can be checked at compile time.
  - Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and then insert the necessary casts behind the scenes.

Introduction

- The example below (modified to use generics) is a modification of the example on the previous page.

WithGenerics.java

```
 1. package examples.generics;
 2. import java.util.*;
 3.
 4. public class WithGenerics {
 5.     private static int length(Collection<String> c) {
 6.         Iterator<String> i = c.iterator();
 7.         int sum = 0;
 8.         while(i.hasNext()){
 9.             sum += i.next().length();
10.         }
11.         return sum;
12.     }
13. }
```

- The code `<DataType>` is typically read as "of *DataType*."
  - Therefore, the declarations above would typically be read as:
    - "a `Collection` of `String` `c;`" and
    - "an `Iterator` of `String` `i`."
- Code using generics is clearer and safer.
  - An unsafe cast and a number of extra parentheses have been eliminated.
  - The fact that the `length` method has to be a `Collection` of `String` objects has become part of the method's signature.
    - The compiler can verify at compile time that the type constraints are not violated at run time.
    - It can now be stated with certainty that it will not throw a `ClassCastException` at run time.

Defining Simple Generics

- Small excerpts from the `Collection` and `Iterator` interfaces are shown below.

```
public interface Collection<E>{
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E>{
    E next();
}
```

- The angle brackets and their contents are the declarations of the "formal type parameters" of the interfaces.
  - With a few exceptions, these type parameters can be used throughout the generic declaration of the class or interface, where you would use ordinary types.

- In the introduction, there was an invocation of the generic type declaration `Collection<String>`.
  - In the invocation, all occurrences of the formal type parameter (`E` in this case from the definition for Collection above) are replaced by the actual type argument (`String`).
  - The choice of the type parameter being named `E` is arbitrary, although it is recommended that it be a single uppercase character to distinguish it from normal classes and interfaces.

- When a generic declaration is invoked (much like when a method is invoked and actual values are substituted for the parameters), the actual type arguments are substituted for the formal type parameters.

Generics and Subtyping

- While it is true that the `String` class extends the `Object` class, allowing a `String` to be stored in a reference to an `Object`, the same does not hold true for generic types.
  - An `ArrayList<String>` would not be allowed to be stored in a reference to an `ArrayList<Object>`.
  - An example of this can be seen in the two lines of code below.
    - Whereas the first line will compile, the second line would result in a compiler error.

```
Object o = new String("");
ArrayList<Object> list = new ArrayList<String>();
```

  - In general, if `ClassA` is a subclass of, or implements, `ClassB`, and `G` is some generic type declaration, it is NOT the case that `G<ClassA>` is a subtype of `G<ClassB>`.
- Although the rules so far have been quite restrictive, the pages that follow examine more flexible generic types.

Wildcards

- In a version of Java prior to Java SE 5, a method that prints out the elements of a collection may look something like the code below.

```
public static void print(Collection c){
    Iterator iterator = c.iterator();
    for( int i = 0; i < c.size(); i++){
        System.out.println(iterator.next());
    }
}
```

- Below is one attempt at rewriting the method using generics.

```
public static void print(Collection<Object> c){
    Iterator<Object> iterator = c.iterator();
    for( int i = 0; i < c.size(); i++){
        System.out.println(iterator.next());
    }
}
```

  - The problem is that although the second example uses generics, it is not as useful as the first example.
    - The first example could be passed any type of `Collection` as a parameter, whereas the second example only takes `Collection<Object>`.
    - As explained on the previous page, `Collection<Object>` is not a supertype of all kinds of collections.

- The next page describes, through the use of a wildcard, how to declare a generic type that is a supertype to all kinds of collections.

Wildcards

- `Collection<?>`
  - The above syntax, specifically the "`?`" inside the angle brackets, is used to define a generic type whose element type matches anything.
  - Referred to as a wildcard type, it is often called a "Collection of Unknown."

- Rewriting the method from the previous page using generics and a wildcard results in the following.

```
public static void print(Collection<?> c){
    Iterator<?> iterator = c.iterator();
    for( int i = 0; i < c.size(); i++){
        System.out.println(iterator.next());
    }
}
```

- In the above code, it is always safe to call methods that retrieve information from the collection, such as the `iterator` and `size` methods on `c`.
  - It would not compile if inside the `print` method the `add` method were called.
    - Since the actual element type represented by `c` is not known, it is impossible to add objects to it.
    - The exception to this would be the `null` object, which is always a member of every type.

- Through the use of a wildcard, we have seen how to declare a generic type that is a supertype to all kinds of collections.
  - Often, a generic type is desired that is a supertype to a generic subtype.

Bounded Wildcards

- Consider a class named `Person` with a `getName` method.
  - Now consider several subclasses of `Person`, such as `Employee` and `Customer`.
  - It is desirable to define a method that takes as a parameter a collection of such objects and iterate through them.
  - As shown earlier, one might try (unsuccessfully) to accomplish this with code similar to this.

```
public static void printNames(Collection<Person> c){
    Iterator<Person> iterator = c.iterator();
    for( int i = 0; i < c.size(); i++){
        System.out.println(iterator.next().getName());
    }
}
```

    - The problem with the above code is that the `printNames` method can only accept a collection of `Person` objects, not the `Employee` or `Customer` objects.

- `Collection<? extends Person>` is an example of a bounded wildcard.
  - The `?` stands for an unknown type, just like the wildcards we saw earlier.
  - However, in this case, we know that this unknown type is a subtype of `Person`.
  - We say that `Person` is the upper bound of the wildcard.
  - The use of the bounded wildcard above would allow a collection of `Person`, `Employee`, or `Customer` objects to be passed to the `printNames` method.

Generic Methods

- The `printNames` method could be written using bounded wildcards as shown below.

```
public static void printNames(
        Collection<? extends Person> c){

    Iterator<? extends Person> iterator =
            c.iterator();
    for( int i = 0; i < c.size(); i++){
        System.out.println(iterator.next().getName());

    }
}
```

- This method could also be rewritten as a generic method by declaring a type variable to express the upper bound imposed by the wildcard as shown below.

```
public static <P extends Person> void printNames(
        Collection<P> c){

    Iterator<P> iterator = c.iterator();
    for( int i = 0; i < c.size(); i++){
        System.out.println(iterator.next().getName());

    }
}
```

- The generic version of the `printNames` method:
  - is no simpler than the wildcard version; and
  - the declaration of the type variable does not gain us anything.
    - In a case like this, the wildcard solution is typically preferred over the generic solution.
- Generic methods are required to express a relationship between two parameters or between a parameter and a return value.

Exercises

1. Modify the `TestProductDBFromFile` application created earlier to use generics.

Chapter 8:
Advanced I/O

1) Introduction 8-2

2) Basic File I/O Example 8-4

3) Buffered I/O 8-5

4) The `Console` Class 8-8

5) Object Serialization 8-10

6) Serialization Issues 8-16

7) Compressed Files 8-18

8) Zip File Example 8-19

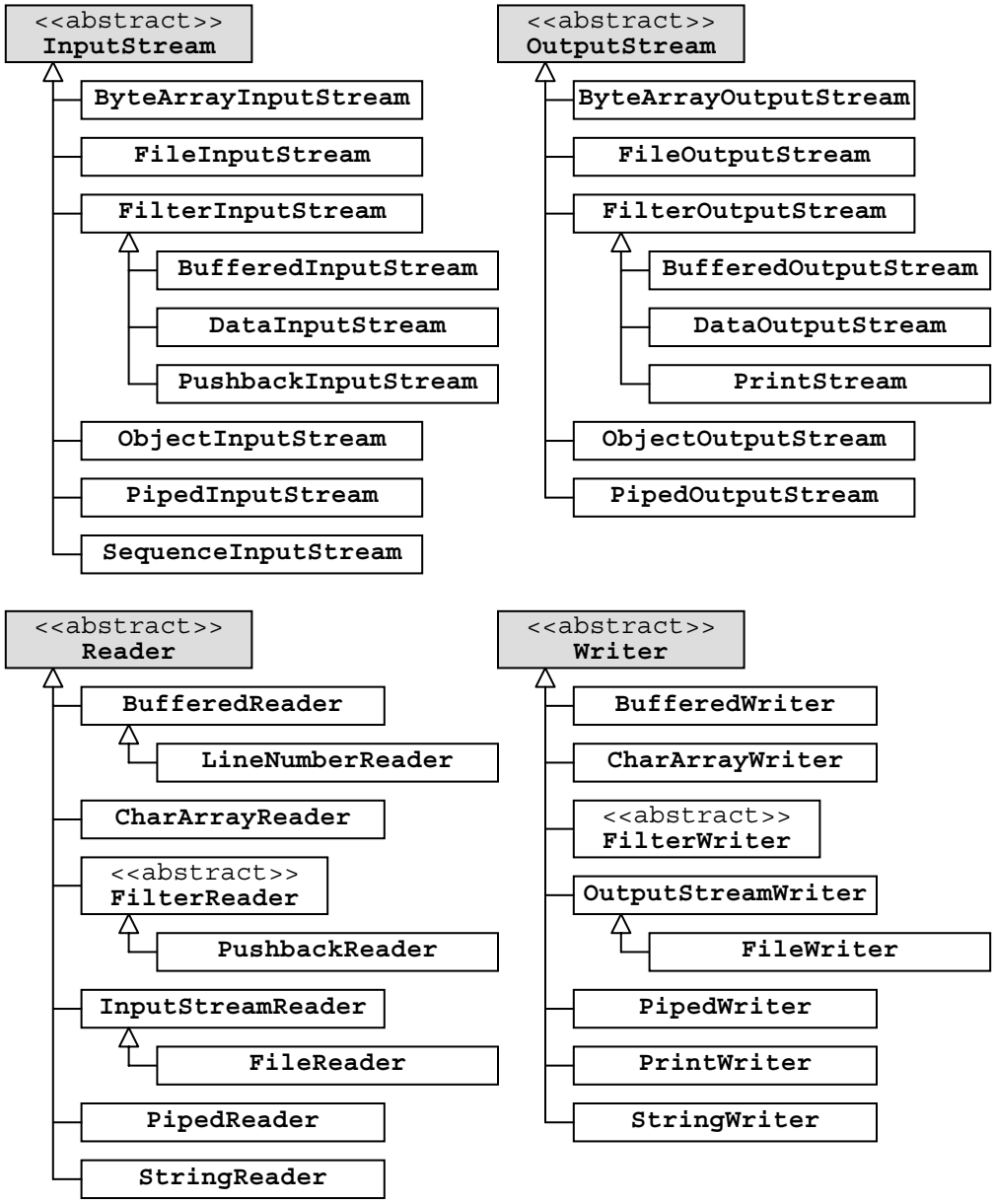
9) Writing Your Own I/O Classes 8-21

10) Property Files 8-23

11) The `Preferences` Class 8-26

Introduction

- The `java.io` package contains many classes, which correspond to various ways of performing input and output in Java.
  - Most of the classes in the `java.io` package descend from one of four abstract classes, as shown in the hierarchies below.



Introduction

- Subclasses of `InputStream` and `OutputStream` are referred to as **byte streams**, which are used to perform input and output of 8-bit bytes.

- Subclasses of `Reader` and `Writer` are referred to as **character streams**, which are used to perform input and output of characters (16-bit bytes), automatically handling translation to and from the local character set.

- The `System` class defines three standard streams, accessible as `System.in`, `System.out`, and `System.err`.
  - `System.out` and `System.err` are of type `PrintStream`, which provides the overloaded `print` and `println` methods used throughout the course.
    - While `PrintStream` is technically a byte stream, it utilizes an internal character stream to emulate a character stream.
  - `System.in` is of type `InputStream` and therefore, has the following methods.
    - The `read` method returns the next byte of data (as an `int`) from the stream each time it is called or a `-1` indicating the end of the stream has been reached.
    - The `available` method returns the number of bytes available from the stream.
    - The `close` method closes the stream and releases any associated system resources.

- The example on the following page demonstrates the similarities between byte streams and character streams.

Basic File I/O Example

FileCopy.java

```
 1. package examples.io;
 2. import java.io.*;
 3. public class FileCopy {
 4.     public static void main(String args[])
 5.             throws IOException {
 6.         String prompt = "Indicate the type of file"
 7.                 + "\n(1) Binary File\n(2)Text File";
 8.         System.out.println(prompt);
 9.         char choice = (char) System.in.read();
10.         if (choice == '1') {
11.             FileInputStream input =
12.                 new FileInputStream(args[0]);
13.             FileOutputStream output =
14.                 new FileOutputStream(args[1]);
15.             copy(input, output);
16.             input.close();
17.             output.close();
18.         } else if (choice == '2') {
19.             FileReader input =
20.                 new FileReader(args[0]);
21.             FileWriter output =
22.                 new FileWriter(args[1]);
23.             copy(input, output);
24.             input.close();
25.             output.close();
26.         }
27.     }
28.
29.     private static void copy(InputStream input,
30.             OutputStream output) throws IOException {
31.         int data;
32.         while ((data = input.read()) != -1)
33.             output.write(data);
34.     }
35.
36.     private static void copy(Reader input,
37.             Writer output) throws IOException {
38.         int data;
39.         while ((data = input.read()) != -1)
40.             output.write(data);
41.     }
42. }
```

Buffered I/O

- The example on the previous page demonstrated the reading of data from the keyboard and the use of the `read` and `write` methods to perform either:
  - byte based I/O using subclasses of `InputStream` and `OutputStream`, or
  - character based I/O using subclasses of `Reader` and `Writer`
  - The example also demonstrated the use of the `close` method when I/O was complete.

- The un-buffered I/O performed in the previous example results in each read and write request being handled directly by the underlying operating system.
  - This type of I/O is inefficient, since each request may involve the overhead of disk access or network activity.

- To reduce the overhead associated with unbuffered I/O, the `java.io` package includes classes that implement buffered I/O streams.
  - These buffered streams use a technique that allows two or more classes to work together.
    - This technique is often referred to as "wrapping" one stream within another stream.

- One such buffered I/O class is the `BufferedReader` class.
  - This class defines a `readLine` method that buffers what it reads until it reaches the end of a line and returns the buffer as a `String`.

Buffered I/O

- The example that follows constructs a `FileReader` object, and then "wraps" it in a `BufferedReader` in order to read a file line-by-line.
  - The constructor for `BufferedReader` takes any subclass of `Reader` as its parameter.
- In order to obtain the name of the file to read from, the example also prompts for user input from the keyboard.
  - Although possible to read a byte at a time from the keyboard (as demonstrated in the previous example), we wish to use the `readLine` method of a `BufferedReader` for keyboard input also.
    - Since `System.in` is of type `InputStream`, it cannot be passed to the constructor of a `BufferedReader` that expects a `Reader`.
    - The `InputStreamReader` class is a class that essentially converts an `InputStream` to a Reader.
- The example outputs the text being read to both a `File` and the screen (`System.out`).
  - A `PrintWriter` will be used for the output to a File.
    - Since a `PrintWriter` buffers its output, it also "wraps" another `Writer` to accomplish this.
    - As a convenience, with the introduction of Java SE 5, there are now constructors that accept a `File` or a `String` (representing the file name).
    - Previous releases required first constructing a `FileWriter` and then passing it to the constructor of a `PrintWriter`.

Buffered I/O

BufferedFileCopy.java

```
 1. package examples.io;
 2. import java.io.*;
 3. public class BufferedFileCopy {
 4.     public static void main(String args[]){
 5.         BufferedReader br = null;
 6.         PrintWriter pw = null;
 7.         BufferedReader kb = new BufferedReader(
 8.                 new InputStreamReader(System.in));
 9.         String prompt = "Name of the file to ";
10.         System.out.println(prompt + "read from:");
11.         String fromFile, toFile;
12.         try {
13.             fromFile = kb.readLine();
14.             System.out.println(prompt + "write to:");
15.             toFile = kb.readLine();
16.         } catch (IOException e) {
17.             System.err.println("Can't read from kb");
18.             return;
19.         }
20.         try {
21.             br = new BufferedReader(
22.                     new FileReader(fromFile));
23.             pw = new PrintWriter(toFile);
24.             String theLine;
25.             while((theLine = br.readLine())!= null){
26.                 System.out.println(theLine);
27.                 pw.println(theLine);
28.             }
29.         }catch (IOException e) {
30.             e.printStackTrace();
31.         }finally{
32.             try {
33.                 if(br != null)
34.                     br.close();
35.             } catch (IOException e) {
36.                 e.printStackTrace();
37.             }
38.             pw.close();
39.         }
40.     }
41. }
```

The `Console` Class

- Obtaining user input from the console during runtime of a Java application has often been accomplished by printing a prompt to the screen followed by reading input from the keyboard as shown in the snippet of code below (taken from the previous example).

```
BufferedReader kb = new BufferedReader(
                new InputStreamReader(System.in));
String prompt = "Name of the file to ";
System.out.println(prompt + "read from:");
String fromFile = kb.readLine();
```

- The `Console` class (introduced in Java SE 6) simplifies the above process dramatically.
  - An instance of this class can be obtained by invoking the `System.console()` method.
  - The `readLine` method of `Console` takes a formatted `String` parameter as the prompt, and returns the user input as a `String`.
  - The `readPassword` method takes a formatted `String` parameter as the prompt but does not echo the user input to the screen, and returns the user input as a `char` array.
    - Returning the data in a `char` array is done for security purposes.
    - Since a `String` is immutable, it cannot be changed (i.e., cleared in this case), and as such has potential of remaining in system memory until overwritten.
    - Keep in mind that modern computers often rely on virtual memory (memory written the hard drive), so technically the data may be accessible long after a program terminates.
    - A `char` array, on the other hand, can have each of its primitive `char` elements overwritten (i.e. cleared) prior to being made available for garbage collection.

The `Console` Class

- The example below demonstrates the use of the `Console` class to obtain user input.

ConsoleTest.java

```
 1. package examples.io;
 2.
 3. import java.io.Console;
 4. import java.util.Arrays;
 5.
 6. public class ConsoleTest {
 7.     public static void main(String[] args) {
 8.         Console c = System.console();
 9.         if(c == null){
10.             System.err.println("No Console...");
11.             System.exit(1);
12.         }
13.         String name = c.readLine("User Name:");
14.         char [] pwd = c.readPassword("Password:");
15.         //Code to process password would be here
16.
17.         //Manually zero out password array
18.         Arrays.fill(pwd, ' ');
19.     }
20. }
```

- Note that, depending on the environment, the console may not always be available.
  - For this reason, it is always a good idea to test that the reference returned from `System.console()` is not `null`.
- Another way to obtain user input is the use of the following statement code.
  - The statement relies upon a utility function of the SWING API to show a graphical interface for user input, and return the result as a `String`.

```
String resp = JOptionPane.showInputDialog("prompt");
```

Object Serialization

- Many Java technologies, such as networking, often need to pass and receive Java objects.
  - A Client/Server application may have the client package an object and send it over a network connection to be processed by the server.
- The easiest way of reading and writing objects is to have these objects implement the `Serializable` interface.
  - The `Serializable` interface has no methods.
    - This type of interface is often referred to as "**marker interface.**"
  - Its purpose is to force the programmer to give permission for a class to be serialized.
  - A `Serializable` class can have persistent objects.
    - If what is contained in the implementation of `Serializable`, is not `Serializable` itself, then a `NotSerializableException` will be thrown at runtime.
- Serialization relies upon two main I/O classes.
  - `ObjectOutputStream`
    - Takes an `OutputStream` in its constructor and provides a `writeObject()` method
  - `ObjectInputStream`
    - Takes an `InputStream` in its constructor and provides a `readObject()` method

Object Serialization

- The example below defines a class named `Contact`, which implements `Serializable`.
  - In order for the `Contact` class to truly be `Serializable`, all of its instance data must also be `Serializable`.
    - The `String` class already implements `Serializable`.
    - Primitive data types in Java are always `Serializable`.

Contact.java

```
 1. package examples.io;
 2.
 3. import java.io.Serializable;
 4.
 5. public class Contact implements Serializable{
 6.     private String name, city, state;
 7.     private int zip;
 8.
 9.     public Contact(String name, String city,
10.             String state, int zipCode) {
11.         this.name = name;
12.         this.city = city;
13.         this.state = state;
14.         this.zip = zipCode;
15.     }
16.
17.     public String toString(){
18.         StringBuilder sb = new StringBuilder(100);
19.         sb.append(name).append('\n');
20.         sb.append(city).append(", ").append(state);
21.         sb.append('\n').append(zip).append('\n');
22.         return sb.toString();
23.     }
24. }
```

ObjectOutputStream

- The application below uses an `ObjectOutputStream` to serialize an `ArrayList` of `Contact` objects in two different ways.
  - The application will first serialize each of the individual `Contact` objects to a single file.
  - Then, the application will serialize the `ArrayList` object itself to a different file.
    - Note that this is only possible since `ArrayList` implements `Serializable`.

WriteContacts.java

```
 1. package examples.io;
 2.
 3. import java.util.ArrayList;
 4. import java.io.*;
 5.
 6. public class WriteContacts {
 7.     static ArrayList<Contact> list;
 8.     public static void main(String[] args) {
 9.         getContacts();
10.         persistContacts();
11.         persistList();
12.     }
13.
14.     public static void getContacts(){
15.         list = new ArrayList<Contact>();
16.         list.add(new Contact("Joe", "Columbia", "MD",
17.                 21046));
18.         list.add(new Contact("Ann", "Bowie", "MD",
19.                 20715));
20.         list.add(new Contact("Bob", "Fairfax", "VA",
21.                 20151));
22.     }
23.
```

ObjectOutputStream

WriteContacts.java *continued*

```
24.     public static void persistContacts(){
25.         FileOutputStream fos = null;
26.         ObjectOutputStream oos = null;
27.         String fileName = "contacts.ser";
28.         try {
29.             fos = new FileOutputStream(fileName);
30.             oos = new ObjectOutputStream(fos);
31.             for(Contact contact:list)
32.                 oos.writeObject(contact);
33.         } catch (IOException e) {
34.             e.printStackTrace();
35.         } finally{
36.             closeOutput(oos);
37.         }
38.     }
39.
40.     public static void persistList(){
41.         FileOutputStream fos = null;
42.         ObjectOutputStream oos = null;
43.         String fileName = "list.ser";
44.         try {
45.             fos = new FileOutputStream(fileName);
46.             oos = new ObjectOutputStream(fos);
47.             oos.writeObject(list);
48.         } catch (IOException e) {
49.             e.printStackTrace();
50.         } finally{
51.             closeOutput(oos);
52.         }
53.     }
54.
55.     public static void closeOutput(OutputStream os){
56.         if(os != null){
57.             try{
58.                 os.close();
59.             }catch(IOException ioe){
60.                 ioe.printStackTrace();
61.             }
62.         }
63.     }
64. }
```

ObjectInputStream

- The application below uses an `ObjectInputStream` to read the serialized data back into memory.
  - The `readContacts` method is written without knowledge of the number of `Contact` objects that were persisted.
    - The `readObject` method will throw an `EOFException` to indicate that the end of the stream has been reached.
  - The `readList` method is written with knowledge that a single object was persisted.

ReadContacts.java

```
 1. package examples.io;
 2. import java.util.ArrayList;
 3. import java.io.*;
 4. public class ReadContacts {
 5.     public static void main(String[] args) {
 6.         ArrayList<Contact> contacts = readContacts();
 7.         for(Contact contact:contacts)
 8.             System.out.println(contact);
 9.         System.out.println("--------------------");
10.         contacts = readList();
11.         for(Contact contact:contacts)
12.             System.out.println(contact);
13.     }
14.     public static ArrayList<Contact> readContacts(){
15.         ArrayList<Contact> list =
16.             new ArrayList<Contact>();
17.         FileInputStream fis = null;
18.         ObjectInputStream ois = null;
19.         String fileName = "contacts.ser";
20.         try {
21.             fis = new FileInputStream(fileName);
22.             ois = new ObjectInputStream(fis);
23.             while(true)
24.                 list.add((Contact)ois.readObject());
25.         } catch (EOFException e) {
26.             //quietly ignore as this is expected
27.         } catch (Exception e) {
```

ObjectInputStream

ReadContacts.java - continued

```
28.             e.printStackTrace();
29.         } finally{
30.             closeInput(ois);
31.         }
32.         return list;
33.     }
34.     public static ArrayList<Contact> readList(){
35.         ArrayList<Contact> list = null;
36.         FileInputStream fis = null;
37.         ObjectInputStream ois = null;
38.         try {
39.             fis = new FileInputStream("list.ser");
40.             ois = new ObjectInputStream(fis);
41.             list =
42.                 (ArrayList<Contact>)ois.readObject();
43.         } catch (Exception e) {
44.             e.printStackTrace();
45.         } finally{
46.             closeInput(ois);
47.         }
48.         return list;
49.     }
50.
51.     public static void closeInput(InputStream is){
52.         if(is != null){
53.             try{
54.                 is.close();
55.             }catch(IOException ioe){
56.                 ioe.printStackTrace();
57.             }
58.         }
59.     }
60. }
```

Serialization Issues

- It is important to note that in the previous application, even though generics is used within the application, the `readObject` method still returns an `Object`.
  - This is because the implementation of generics in Java uses "**type erasure.**"
  - The type parameter information is removed by the compiler and replaced with the appropriate casts.
  - The compile will issue a warning indicating that the runtime has no way of verifying that the object you read is an `ArrayList<Contact>`.
  - Because the type parameter information is erased, the best it can do is check that what you have is an `ArrayList`.

- When an object is serialized, some administrative information is also written to the stream.
  - This information includes the **S**tream **U**nique **ID**entifier (**SUID**), a primitive `long` identifying the version of the class being serialized.
  - The SUID is a 64-bit hash of the class name, interface class names, methods, and fields.
    - It is computed using the signature of a stream of bytes that reflect the class definition.

- The `serialver` tool can be used to calculate and display the SUID of a class specified on the command line.
  - Be sure to give the full class name including package name, (e.g. `serialver java.lang.String`).
  - A graphical version of the tool can be run with the command:

    ```
    serialver -show
    ```

Serialization Issues

- When an object is de-serialized, if the SUID saved with the object does not match the SUID of the version of the class found at runtime, an `InvalidClassException` is thrown.

- If you make a compatible change to a serialized class, you may still want to continue to deserialize previously saved classes without throwing an exception.
  - Compatible changes are those that do not change the data or the public method signatures.
  - To do this, take the SUID of the original class and declare a `static final long` variable named `serialVersionUID` with the same value.
    - To find the SUID value of the original class, you can use the `serialver` tool.

- The final topic needing discussion on object serialization is protecting sensitive information.
  - The easiest way is to declare sensitive data items as `transient`.
    - Note that `transient` and `static` fields are not serialized.

- Another way to secure the serialized objects is to encrypt the stream.
  - This is one of the reasons for using streams to implement serialized objects.
    - (See the documentation for the `javax.crypto` package.)

Compressed Files

- If you need to send files or large objects over the network, you can use compression from inside your Java program using one of several classes.

```
java.util.zip.ZipOutputStream
java.util.zip.GZIPOutputStream
java.util.jar.JarOutputStream
```

- The following example demonstrates the use of `ZipOutputStream` to zip the contents of a directory.
  - The application's `main` method relies on several static helper methods to make the tasks involved easier to follow.
    - The `getUserInput` method will prompt the user for the name of a directory to compress and the name of the zip file to place the directory contents into.
    - The `initializeStreams` method will then instantiate the necessary `ZipOutputStream` from the information obtained through the `getUserInput` method.
    - The `processDirectory` method is a recursive method that will either call itself to process a sub-directory, or call the `zipTheFile` method to zip each file.
    - The `zipTheFile` method is where the actual I/O of the application occurs.
    - Once all of the files have been zipped, the `closeStreams` method will close the `ZipOutputStream`.
  - Portions of the code for the application are detailed on the following pages, while the complete code can be found in the file named `CompressDirectory.java`.

Zip File Example

- The `processDirectory` method of the application is shown below.
  - The method is a recursive method call in that, under certain condition(s), the method calls into itself recursively.
    - The method is used to process the subdirectories of a given directory passed as a parameter to the method.

CompressDirectory.java - *partial listing*

```
 1. public static void processDirectory(File dir,
 2.          ZipOutputStream zipFile){
 3.      String entries[] = dir.list();
 4.      File file = null;
 5.      for (String fName:entries) {
 6.          file = new File(dir, fName);
 7.          if (file.isDirectory())
 8.              processDirectory(file, zipFile);
 9.          else{
10.              zipTheFile(file, zipFile);
11.          }
12.      }
13. }
```

- The code below is used to initialize the `ZipOutputStream`.

CompressDirectory.java - *partial listing*

```
 1. public static boolean initializeStreams(){
 2.      try {
 3.          fos = new FileOutputStream(sourceFile);
 4.      } catch (FileNotFoundException e) {
 5.          e.printStackTrace();
 6.          return false;
 7.      }
 8.      zipOut = new ZipOutputStream(fos);
 9.      return true;
10.  }
```

Zip File Example

- The code that zips the individual files is shown below.
  - A `ZipEntry` object represents each file that is placed in the `ZipOutputStream`.

CompressDirectory.java - *partial listing*

```
 1. public static void zipTheFile(File file,
 2.         ZipOutputStream zipFile){
 3.     byte[] buffer = new byte[4096];
 4.     int bytes;
 5.     FileInputStream in = null;
 6.     try {
 7.         in = new FileInputStream(file);
 8.         System.out.println("Zipping:" + file);
 9.         ZipEntry entry =
10.             new ZipEntry(file.getPath());
11.         zipFile.putNextEntry(entry);
12.         while ((bytes = in.read(buffer)) != -1) {
13.             zipFile.write(buffer, 0, bytes);
14.         }
15.     } catch (IOException e) {
16.         e.printStackTrace();
17.     } finally {
18.         if(in != null){
19.             try{
20.                 in.close();
21.             }catch (IOException ioe){
22.                 ioe.printStackTrace();
23.             }
24.         }
25.     }
26. }
```

Writing Your Own I/O Classes

- The example below extends `FilterReader` in order to design a `Reader` that converts data read to uppercase.
  - Any subclass of `FilterReader` should override the two `read` methods of the class.
  - As a convenience, the class below provides additional constructors.

CustomReader.java

```
 1. package examples.io;
 2. import java.io.*;
 3. public class CustomReader extends FilterReader {
 4.
 5.     public CustomReader(File f)
 6.             throws FileNotFoundException {
 7.         this(new FileReader(f));
 8.     }
 9.     public CustomReader(String s)
10.             throws FileNotFoundException {
11.         this(new File(s));
12.     }
13.     public CustomReader(Reader in) {
14.         super(in);
15.     }
16.     public int read(char[] ary, int offset, int len)
17.             throws IOException {
18.         int cnt = super.read(ary, offset, len);
19.         if (cnt == -1)
20.             return -1;
21.         for (int i = offset; i < offset + cnt; i++) {
22.             ary[i] = Character.toUpperCase(ary[i]);
23.         }
24.         return cnt;
25.     }
26.     public int read() throws IOException {
27.         char[] buf = new char[1];
28.         int result = read(buf, 0, 1);
29.         return result == -1 ? -1 : buf[0];
30.     }
31. }
```

Writing Your Own I/O Classes

- The application below tests the `CustomReader` defined on the previous page.

CustomReaderTest.java

```
 1. package examples.io;
 2. import java.io.*;
 3.
 4. public class CustomReaderTest {
 5.     public static void main(String args[])
 6.             throws IOException {
 7.         CustomReader cr = new CustomReader(args[0]);
 8.         BufferedReader br = new BufferedReader(cr);
 9.         String theLine;
10.         while ((theLine = br.readLine()) != null) {
11.             System.out.println(theLine);
12.         }
13.         br.close();
14.     }
15. }
```

Property Files

- Robust applications usually have customization parameters with which to be concerned.
  - These files are typically referred to as configuration and/or initialization files.
  - Some examples of these are listed below.
    - `.exrc` files used by the vi editor in a UNIX platform
    - `.ini` files used by many applications on all major platforms
- Applications use the application-specific files that are usually read when the application is launched and written when the application is about to terminate.
  - During the running of the application, a means is typically provided so that a user could enter new values for one or more custom features, which could then be applied when the application is next launched.
- Java applications (and other Java code bodies) are able to represent this type of information through the use of a `Properties` object.
  - The `Properties` class extends `Hashtable` and provides several methods to `store` and `load` properties.
    - `Properties` are managed as key/value pairs.
    - In each pair, the key and value are both `String` values.
- The code on the following page creates a utility class for the reading and writing of a properties file that can then be used within an application.

The `Properties` Class

ConfigUtil.java

```
 1. package examples.io;
 2.
 3. import java.io.*;
 4. import java.util.Properties;
 5.
 6. public class ConfigUtil {
 7.     static Properties p;
 8.     static String cnfgFile = "config.properties";
 9.     static {
10.         p = new Properties();
11.         p.put("time.format", "ampm");
12.         p.put("date.format", "short");
13.         p.put("user", "New User");
14.         File f = new File(cnfgFile);
15.         if(f.exists()){
16.             FileInputStream fis = null;
17.             try {
18.                 fis = new FileInputStream(cnfgFile);
19.                 p.load(fis);
20.                 fis.close();
21.             } catch (IOException ioe) {
22.                 ioe.printStackTrace();
23.             }
24.         }
25.     }
26.     public static void persist() throws IOException{
27.         FileOutputStream fos =
28.             new FileOutputStream(cnfgFile);
29.         p.store(fos, "Sample Config File");
30.         fos.close();
31.     }
32.     public static Properties getConfiguration(){
33.         return p;
34.     }
35. }
```

- The static block above ensures that if the `config.properties` file does not exist, the `Properties` object will use a set of default values.
  - This happens when the program is run for the first time, or it the user deletes the properties file.

The `Properties` Class

- The application below tests the `ConfigUtil` class.

ConfigTester.java

```
 1. package examples.io;
 2.
 3. import java.io.IOException;
 4. import java.util.*;
 5.
 6. public class ConfigTester {
 7.     public static void main(String args[])
 8.             throws IOException {
 9.         Properties p = ConfigUtil.getConfiguration();
10.         outputData(p);
11.         p.put("user", "Joe Smith");
12.         p.put("time.format", "%tT");
13.         p.put("date.format", "%tc");
14.         System.out.println("---------------");
15.         outputData(p);
16.         ConfigUtil.persist();
17.     }
18.     public static void outputData(Properties p){
19.         System.out.println(p.getProperty("user"));
20.         String tFrmt = p.getProperty("time.format");
21.         String dFrmt = p.getProperty("date.format");
22.         Calendar c = Calendar.getInstance();
23.         System.out.printf(tFrmt + "%n", c);
24.         System.out.printf(dFrmt + "%n", c);
25.     }
26. }
```

- The output from the above application is shown below for two consecutive runs.

First Run Output	Second Run Output:
New User 04:07:48 PM 02/13/08 --------------- Joe Smith 16:07:48 Wed Feb 13 16:07:48 EST 2008	Joe Smith 16:08:22 Wed Feb 13 16:08:22 EST 2008 --------------- Joe Smith 16:08:22 Wed Feb 13 16:08:22 EST 2008

The `Preferences` Class

- The `java.util.prefs.Preferences` class (introduced in Java SE 1.4) is a hierarchical collection of nodes representing preference data.
  - `Preferences` allow applications to store and retrieve user and system preference and configuration data.
    - This data is stored persistently in an implementation-dependent backing store.
    - Typical implementations include flat files, OS-specific registries, directory servers, and SQL databases.
    - The user of this class does not need to be concerned with details of the backing store.

- There are two separate trees of preference nodes.
  - one for user preferences where each user has a separate user preference tree
  - one for system preferences where all users in a given system share the same system preference tree

- Nodes in a preference tree are named in a similar fashion to directories in a hierarchical file system.
  - Every node in a preference tree has the following.
    - a node name (which is not necessarily unique)
    - a unique absolute path name
    - a path name relative to each ancestor including itself
  - The root node has a node name of the empty string (`""`).
    - Every other node has an arbitrary node name, specified at the time it is created.

The `Preferences` Class

- Obtaining a reference to a `Preferences` object is done using one of the following `static` methods of the class.
  - `Preferences systemNodeForPackage(Class c);`
    - Returns the system preference node that is associated with the specified class's package
  - `Preferences systemRoot();`
    - Returns the root preference node for the system
  - `Preferences userNodeForPackage(Class c);`
    - Returns the user preference node that is associated with the specified class's package
  - `Preferences userRoot();`
    - Returns the root preference node for the user
- Key/value pairs are handled in a similar way to how they are handled with the `Properties` class.
  - The `get` methods of the class, take two parameters.
    - The first parameter represents the key with which the value is associated.
    - The second parameter is the default value to return if the key is not found or the backing store is inaccessible.
- The two applications on the next page demonstrate how easy it is to begin using the `Preferences`.
  - Each application demonstrates a different technique for obtaining a reference to the desired preference node.

Preferences Example

StorePreferences.java

```
 1. package examples.io;
 2. import java.util.prefs.Preferences;
 3. public class StorePreferences {
 4.     public static void main(String[] args){
 5.         Class c = StorePreferences.class;
 6.         Preferences userPrefs =
 7.             Preferences.userNodeForPackage(c);
 8.         userPrefs.put("color", "blue");
 9.         Preferences sysPrefs =
10.             Preferences.systemNodeForPackage(c);
11.         sysPrefs.putInt("portNumber", 8080);
12.     }
13. }
```

ShowPreferences.java

```
 1. package examples.io;
 2. import java.util.prefs.Preferences;
 3. public class ShowPreferences {
 4.     public static void main(String[] args){
 5.         String nodeName = "/examples/io";
 6.         Preferences userPrefs =
 7.             Preferences.userRoot().node(nodeName);
 8.         String color = userPrefs.get("color", "red");
 9.         System.out.println(color);
10.
11.         Preferences sysPrefs =
12.             Preferences.systemRoot().node(nodeName);
13.         int port = sysPrefs.getInt("portNumber", 80);
14.         System.out.println(port);
15.     }
16. }
```

- Multiple users on the same system could maintain their own "`color`" preference, while the "`portNumber`" system preference would be shared by all users.

Exercises

1. Modify the `ProductDBFromFile` so that the name of the file can be read from a `System` property named "`textfile.name`."

   - A system property can be passed to the JVM when it starts up by using the "`-D`" option. An example of which is shown below.

     ```
     java -DpropName=propValue fully.qualified.class.to.run
     ```

     - ***propName*** would be `textfile.name` and ***propValue*** would be the path and name of the file to read from.

2. Save a copy of the `TestProductDBFromFile.java` application from earlier as `SerializeProductDB.java` and modify it so that, instead of displaying the list of `Product` objects, it serializes each of the objects to a file named on the command line.

3. Modify the `Configuration.java` file supplied in the starters directory to read from the file named `configuration.cfg` in the `data` directory of the labfiles.

   - The `TestConfiguration.java` application in the starters directory is complete and can be used to test your `Configuration` class.

4. Save a copy of `ProductDBFromFile.java` as `ProductDBFromSerFile.java` and modify it in such a way that it populates itself of `Products` from the serialized output created above instead of from a text file.

   - This class should obtain the name of the serialized file from which to read - from a `System` property loaded by the `Configuration` class.

This Page Intentionally Left Blank

Chapter 9:
Java Security

1) Security Overview 9-2
2) Java Language Security 9-4
3) Access Control 9-6
4) Policies and Permissions 9-9
5) Property Permissions 9-10
6) File Permissions 9-11
7) Socket Permissions 9-13
8) Policy Files 9-14
9) Encrypting Important Data 9-17
10) Ciphers 9-20
11) Summary 9-23

Security Overview

- Java includes a large set of APIs and tools that provide developers with a security framework for writing code and a set of tools for securely managing applications.

- The security framework and tools can be broken down into the following categories.
  - Java language security and bytecode verification
    - Automatic memory management, garbage collection, and range-checking of arrays are several of the ways the language itself is secured.
    - Certain levels of security can also be achieved through the use of the `public`, `private`, `protected`, and *default* access levels.
    - The JVM utilizes a bytecode verifier which is invoked automatically to ensure only legitimate code is executed at runtime.
  - Basic security architecture
    - The Java platform provides a number of built-in security providers and supports the installation of custom providers if desired.
    - Configuration of the provider(s) can be customized using security properties, which can be loaded from a file or set dynamically with method calls of the `Security` class.
  - Access Control
    - Permissions can be used to control access to resources.
    - A security policy can be used to manage the permissions.
    - A `SecurityManager` is used to determine which actions are permitted at runtime.

Security Overview

- Cryptography
  - Many of the most commonly used cryptographic algorithms are built-in to the Java platform.
  - Other providers can be plugged in to the cryptography framework.
- Public Key Infrastructure (PKI)
  - PKI enables secure exchange of information based on public key cryptography using digital certificates.
  - Key and certificate stores provide for long-term persistent storage.
  - `keytool` and `jarsigner` are two built-in tools for working with keys, certificates, and key stores.
- Authentication
  - APIs are provided to enable user authentication via pluggable login modules.
- Secure Communication
  - The Secure Socket Layer (SSL) protocol implemented by Java can be used to provide secure transmission of data.

- This chapter introduces some of the APIs and tools that provide developers with a security framework for their Java applications.
  - Specifically, this chapter will look at how to use a security manager to control access to system resources.
  - It will also show how data can be protected through encryption.

Java Language Security

- The JVM relies on a built-in bytecode verifier to ensure only legitimate code is executed at runtime.

- The example that follows demonstrates an attempt to bypass the access level of `private` for a given objects data.

  - `SomeObject` will first be defined as preventing direct access to its instance data.

    - Therefore, the `TestObject` application that accesses `SomeObject` will not compile since it cannot access the data directly.

SomeObject.java

```
1.  package examples.security;
2.  public class SomeObject {
3.      private String someData;
4.  }
```

TestObject.java

```
1.  package examples.security;
2.  public class TestObject {
3.      public static void main(String[] args) {
4.          SomeObject obj = new SomeObject();
5.          obj.someData = "Testing";
6.          System.out.println(obj.someData);
7.      }
8.  }
```

  - The next page describes the steps one might take to attempt to bypass the private access to `SomeData`.

    - Although compilation will succeed, the bytecode verifier will catch it at runtime and not allow the code to run successfully.

Java Language Security

- Modifying the original `SomeData` to the following will permit the `TestObject` class to compile successfully.

SomeObject.java

```
1.  package examples.security;
2.  public class SomeObject {
3.      public String someData;
4.  }
```

  - The above modification allows the `TestObject` class to compile successfully.
  - Now the developer has two `.class` files where the `TestObject` accesses `someData` of `SomeObject` directly.

- Now the modified `SomeObject.class` is replaced with the original while leaving the `TestObject.class` the same.
  - This results in `TestObject` written in such a way that it still accesses the private data of `SomeObject` directly.
  - Attempting to run the TestObject will result in the bytecode verifier catching the illegal code at run time and throwing the following exception:

```
Exception in thread "main"
java.lang.IllegalAccessError: tried to access field
examples.security.SomeObject.someData from class
examples.security.TestObject at
examples.security.TestObject.main(TestObject.java:5)
```

    - Duplicating the above scenario using an IDE such as Eclipse or NetBeans is often difficult if not impossible, since it verifies source code as it is written.
    - Compiling in a shell such as DOS may be required.

Access Control

- A security manager is not automatically used when running a Java application, allowing "*unrestricted access*" to the resources.
  - The term "*unrestricted access*," as used above, implies the JVM places no additional restrictions on the resources.
  - Any platform restrictions placed on a resource, such as a file being read-only, remain in effect, regardless of the whether a security manager is in effect or not.
- In order to understand how access to resources can be controlled within an application, we will create an application that accesses various types of resources.
  - The application will first be run without the use of a security manager, and will be able to perform the following:
    - Read various properties available from the System class.
    - Read and write to and from various files on the local file system.
    - Resolve an IP address and connect to the remote machine.
  - Each resource being accessed is done in its own helper method within the application to make the code easier to follow.
- Once the application is run without a security manager and the output reviewed, the code should be run again supplying the following system property to the JVM when run:

```
-Djava.security.manager
```

  - This will result in the default security manager being used whose results are discussed following the example code.

Access Control

Resources.java

```
 1. package examples.security;
 2.
 3. import java.io.*;
 4. import java.net.*;
 5.
 6. public class Resources {
 7.
 8.     public static void main(String[] args) throws
 9.             Exception {
10.         //Read System Properties
11.         printSysProp("os.name");
12.         printSysProp("user.home");
13.
14.         //Read from various files
15.         URI uri =
16.             Resources.class.getResource(".").toURI();
17.         File path = new File(uri.getPath());
18.         String fileNames [] = { "data.txt",
19.             "../../data/products.txt",
20.             "../../data/nosuchfile.txt"
21.         };
22.         for(String fileName: fileNames){
23.             readFromFile(new File(path, fileName));
24.         }
25.
26.         //Write to a file
27.         writeToFile(new File(path, fileNames[0]));
28.
29.         //Resolve an IP address
30.         resolveIP("trainingetc.com");
31.
32.         //Read web page at that Internet Address
33.         readWebPage("trainingetc.com");
34.     }
35.
36.     private static void printSysProp(String name) {
37.         System.out.print("Property: ");
38.         System.out.print(name + "=");
39.         System.out.println(System.getProperty(name));
40.     }
```

▸ Code continued on next page.

Access Control

Resources.java - continued

```
41.
42.     private static void readFromFile(File f){
43.         if(!f.exists()){
44.             return;
45.         }
46.         FileReader fr = null;
47.         BufferedReader br = null;
48.         try {
49.             fr = new FileReader(f);
50.             br = new BufferedReader(fr);
51.             System.out.println(br.readLine());
52.             br.close();
53.         } catch (IOException e) {
54.             e.printStackTrace();
55.         }
56.     }
57.
58.     private static void writeToFile(File f){
59.         PrintWriter pw;
60.         try {
61.             pw = new PrintWriter(f);
62.             pw.println("Write this data to a file");
63.             pw.close();
64.         } catch (IOException e) {
65.             e.printStackTrace();
66.         }
67.     }
68.
69.     private static void resolveIP(String ip) throws
70.             UnknownHostException{
71.         InetAddress ia =
72.             InetAddress.getByName(ip);
73.         System.out.println(ia);
74.     }
75.
76.     private static void readWebPage(String ip){
77.         String ips [] = {ip};
78.         examples.networking.ReadWebPage.main(ips);
79.     }
80. }
```

Policies and Permissions

- When the application is run with a security manager, it becomes evident that certain resources are restricted by default.
  - The "`os.name`" property was able to be accessed, but accessing "`user.home`" resulted in an `AccessControlException` being generated.

    ```
    java.security.AccessControlException: access denied
    (java.util.PropertyPermission user.home read)
    ```

  - The `SecurityManager` class relies on a default policy file that lists permissions that are allowed and/or restricted.
    - Permission to access certain system properties is restricted by default to prevent certain information about a particular environment from being accessible by a user.
- Since the application did not catch the `AccessControlException`, it was handled automatically by the JVM and the program terminated.
  - Before discussing the default policies and permissions in detail, a different version of the application has been supplied to test the remaining resources that were being accessed.
    - The `DetailedResources.java` (not shown) is supplied in the lab files as a rewrite of `Resources.java`.
    - This file includes a catch block for each of the private methods to handle any `AccessControlException` that may be thrown.
    - Running the `DetailedResources` application with a security manager present will allow all of the methods to be executed in order to get a better understanding of the types of resources that are permitted or denied by the default policy.

Property Permissions

- The classes in the core Java API always have permission to perform any action.
  - All other classes must be given explicit permission to perform sensitive operations when a security manager is in place.
  - Configuring permissions is commonly done through a policy file(s).
    - The policy file will be discussed shortly.
- Property permissions control access to the `System` properties and are controlled by the `java.util.PropertyPermission` class.
- A `PropertyPermission` is composed of a name and a set of actions.
  - The name is the name of the system property being accessed.
    - Names follow the hierarchical property naming convention.
    - An asterisk may appear at the end signifying a wildcard.
  - The actions that can be granted are supplied as a string containing a list one or more comma-separated keywords.
    - The keyword "`read`" allows `System.getProperty` to be called.
    - The keyword "`write`" allows `System.setProperty` to be called.
  - Several examples of the syntax are shown below:

```
// Read the standard java properties
permission java.util.PropertyPermission "java.*", "read";
// Read and write properties from this chapters package
permission java.util.PropertyPermission
        "examples.security.*", "read,write";
```

File Permissions

- A `FilePermission` controls access to a file or directory and is composed of a pathname and a set of actions.
  - Code can always read a file from the directory it was loaded from and its subdirectories.
    - The `examples.security.DetailedResources` class was loaded from the directory that contains the `examples` package (the `bin` directory).
    - Therefore, it can read any file from the `bin` directory or any of its subdirectories - but nothing above the `bin` directory.
  - A pathname is the pathname of the file or directory granted specific actions.
    - A pathname that ends in `/*` indicates all files and directories in that directory but is not recursive into subdirectories.
    - A pathname that ends with `/-` indicates (recursively) all files and directories in that directory.
    - A pathname of `<<ALL FILES>>` matches any file with no restrictions.
    - `${/}` can be used to represent a platform-independent file separator.
    - In general you can use any standard java property of the syntax `${`*`property.name`*`}` such as `${user.home}`
  - The actions that can be granted as part of the permission are supplied as a string containing a list of one or more comma-separated keywords shown below.

    ```
    read    write    execute    delete
    ```

  - Several examples of granting file permissions are shown on the following page.

File Permissions

- Below are several examples of the syntax for granting file permissions.

```
// Allow unrestricted access to all files
permission java.io.FilePermission
    "<<ALL FILES>>", "read,write,delete,execute";


//Allow the users home directory to be read from or
// written to but not subdirectories of users home
permission java.io.FilePermission
    "${user.home}${/}*", "read,write";


//Allow the users home directory to be read from
//  including subdirectories of users home
permission java.io.FilePermission
    "${user.home}${/}-", "read";


//Allow the default temp directory to be read
// from, written to and/or any file within deleted
// (recursively)
permission java.io.FilePermission
    "${java.io.tmpdir}${/}-", "read,write,delete";
```

Socket Permissions

- A `SocketPermission` controls access to a network and is composed of a name and a set of actions.
  - A name is of the form *`hostname:port`*.
    - The *`hostname`* can be specified as a DNS name or a numerical IP address.
    - An asterisk may be included in the leftmost position of a DNS name.
    - The *`port`* number can be a single port number or a range of port of numbers (e.g., `1-1024`).
    - When a range is specified on the port, either side may be omitted.
  - The actions that can be granted are supplied as a string containing a list of one or more comma-separated keywords shown below.

    `accept listen connect resolve`

    - Several examples of granting network permissions are shown below.

```
//Allow all socket operations
permission java.net.SocketPermission
        "*:1-", "accept,listen,connect,resolve";


//Allow http access to all subdomains on apache.org
// (e.g., xml.apache.org, tomcat.apache.org)
permission java.net.SocketPermission
        "*.apache.org:80", "connect,resolve";
```

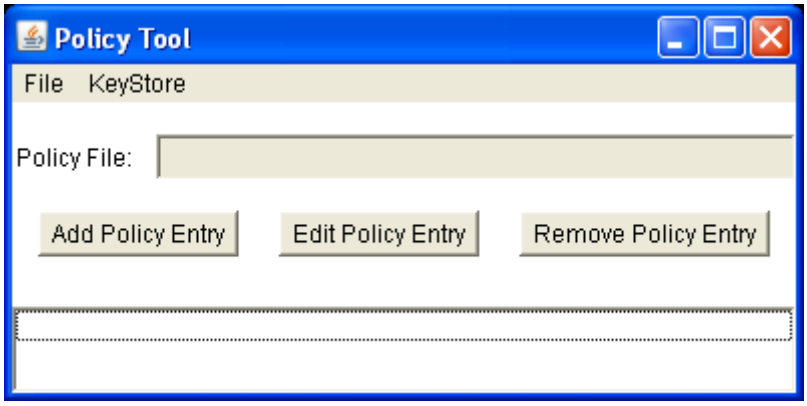
Policy Files

- Permissions are specified in a Java policy file.
  - There is a global policy file named `java.policy` that can be found at the following location on your hard drive:

    `$JREHOME/lib/security/java.policy`

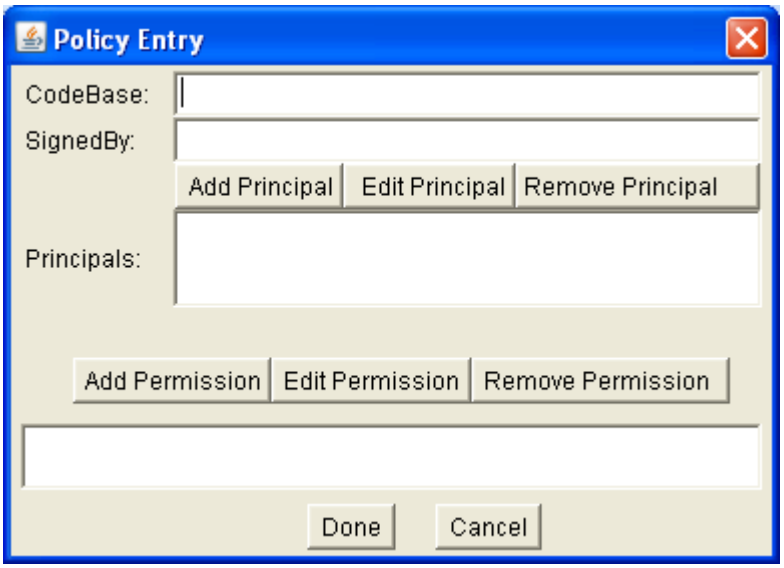
  - In addition, there may exist a policy file called `.java.policy` in the user's home directory.

- A Policy file is a simple text file that can be edited by hand or accessed with the `policytool` that ships with Java.
  - The `policytool` is a graphical application that can be accessed simply by typing in `policytool` from a shell prompt.
  - The user's default policy file does not exist by default. Therefore, when the first time `policytool` is open, the entry for "Policy File:" will be empty.
    - Choose save from the file menu and save it as a file named "`.java.policy`" (filename starts with a "dot"), in your user home directory.
    - If you are not sure what your user home directory is, run the `Resources` application from this chapter without a security manager installed to list the directory in the output.



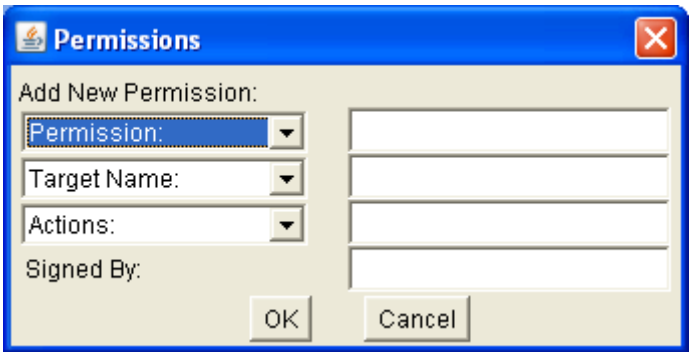
Policy Files

- With the policy file saved, we will now choose to "Add Policy Entry."

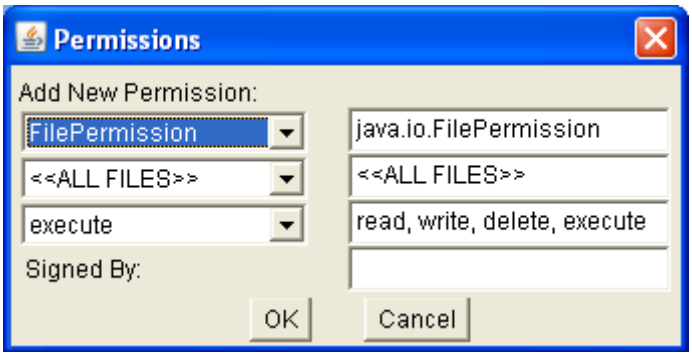


- This chapter focuses on the permissions of the policy file and leaves the codebase and the signing of code for further reading on the student's part if the desire and/or need exists.
  - The "CodeBase:" field allows one to specify from where the class files are loaded that the policy applies to.
  - The "SignedBy" and "Principals" provide a means of digitally signing code with certificates and specifying authorized users of the code.
    - Further details are available in JDK documentation in the `JAVA_HOME/docs/index.html` file.
- For now, we will choose to "Add Permission."

Policy Files



- Using the drop down menus for "Permission:," "Target Name:," and "Actions," add entries as shown below.



- Saving the changes and opening the file created shows the following.

.java.policy

```
1.  /* AUTOMATICALLY GENERATED ON ...*/
2.  /* DO NOT EDIT */
3.
4.  grant {
5.    permission java.io.FilePermission "<<ALL FILES>>",
6.         "read, write, delete, execute";
7.  };
```

- Running `DetailedResources` again with a security manager should now permit the reading and writing of the files.
  - Granting the additional permissions, such that no `AccessControlException` is generated is left as an exercise.

Encrypting Important Data

- The example that follows serializes a `Customer` object.

Customer.java

```
 1. package examples.security;
 2.
 3. import java.io.*;
 4.
 5. public class Customer implements Serializable{
 6.     private String ccNumber;
 7.     private String name;
 8.
 9.     public Customer(String ccNumber, String name) {
10.         this.ccNumber = ccNumber;
11.         this.name = name;
12.     }
13.     public String getCcNumber() {
14.         return "****-****-****-" +
15.             ccNumber.substring(15);
16.     }
17.     public void setCcNumber(String ccNumber) {
18.         this.ccNumber = ccNumber;
19.     }
20.     public String toString(){
21.         return name + ", " + getCcNumber();
22.     }
23.     public static void main(String[] args)
24.             throws Exception {
25.         String number = "0000-1111-2222-3333";
26.         Customer customer =
27.             new Customer(number, "Joe Smith");
28.         Class c = Customer.class;
29.         File path =
30.             new File(c.getResource(".").toURI());
31.         File f = new File(path, "customers.ser");
32.         FileOutputStream fos =
33.             new FileOutputStream(f);
34.         ObjectOutputStream oos =
35.             new ObjectOutputStream(fos);
36.         oos.writeObject(customer);
37.         oos.close();
38.     }
39. }
```

Encrypting Important Data

- Examination of the `customer.ser` file generated from the previous code shows that although the output is not a text file, the credit card number of the customer is clearly visible in the file.
  - This is a security risk that the customers would not be happy with if they were aware their data was being stored in such a way.
  - One way around this would be to mark the `ccNumber` variable in the `Customer` class as `transient`.
    - This would prevent that data from being serialized as part of the default behavior of the `writeObject` method.
    - One problem with this approach though is that the data is now lost when the object is saved to a file.
- What would be convenient is a way in which the data could be stored to the file along with the customer name, but in such a way that the credit card number cannot be determined by looking at the file.
- The first step in this process is to mark the `ccNumber` as `transient`.
- The second step is to have the `Customer` class implement the following methods.

```
private void readObject(ObjectInputStream stream)
     throws IOException, ClassNotFoundException;

private void writeObject(ObjectOutputStream stream)
     throws IOException
```

Encrypting Important Data

- The code below is a snippet of `Customer2.java` that implements the methods as defined on the previous page.

Customer2.java - snippet

```
 1.      private void readObject(ObjectInputStream in)
 2.          throws IOException, ClassNotFoundException {
 3.          // Read all fields that are not transient or
 4.          //   static
 5.          in.defaultReadObject();
 6.          // Read transient data and decrypt
 7.          byte data [] = new byte[24];
 8.          in.read(data, 0, data.length);
 9.          try {
10.              ccNumber = CCEncrypter.decrypt(data);
11.          } catch (Exception exception) {
12.              throw new IOException("Cannot decrypt",
13.                      exception);
14.          }
15.      }
16.      private void writeObject(ObjectOutputStream out)
17.          throws IOException{
18.          // Write all fields that are not transient or
19.          //   static
20.          out.defaultWriteObject();
21.          // Encrypt and write transient data
22.          byte [] data;
23.          try {
24.              data = CCEncrypter.encrypt(ccNumber);
25.              out.write(data, 0, data.length);
26.          } catch (Exception exception) {
27.              throw new IOException("Cannot encrypt",
28.                      exception);
29.          }
30.      }
```

  - The `CCEncrypter` class is where the encrypting and decrypting takes place.
    - Before studying the code for that class, we will briefly discuss the some of the basic concepts of encryption and decryption.

Ciphers

- **Encryption** is the process of taking data (often referred to as clear text) and transforming into what is called cipher text.

- **Decryption** reverses the process by taking the cipher text and transforming back into clear text.

- The mathematical algorithm used for transformation is called a **cipher**.
  - Ciphers generally use keys to encrypt and decrypt data.
    - Symmetric ciphers use symmetric keys (secret keys).
    - Asymmetric ciphers use asymmetric keys (public/private keys).
  - Some of the standard algorithms available in the JDK are:
    - **DES**, **DESede** (Triple DES), **AES**, and **Blowfish**
  - Algorithms are usually combined with padding schemes and modes.
    - Some of the standard modes are **ECB**, **CBC**, **CFB**, **OFB**, and **PCBC**.
    - The standard padding schemes, available as part of the JDK, are **PKCS5Padding** and **NoPadding**.
  - If the mode and padding scheme is not specified they default to an implementation specific value.

- The `javax.crypto.Cipher` class provides an interface to encrypt and decrypt data.

Ciphers

- The `Cipher` class provides several static `getInstance` methods to obtain an instance based on a supplied algorithm.
  - A `Cipher` object is initialized by calling of the overloaded `init()` methods.
    - The first parameter to any of the `init()` methods determines whether the `Cipher` is for encrypting, decrypting, or keywrapping.
    - The second parameter is usually the key to be used.
  - The `update()` methods can be used to process the data to be encrypted or decrypted.
  - The `doFinal()` methods are called to signal that all data has been processed (possibly by the `doFinal()` itself).
- In the example that follows, the `javax.crypto.spec.SecretKeySpec` class will be used to create a `java.security.Key` to be used by the `Cipher`.
  - Since the algorithm used in the example is "`DESede`," also known as "Triple DES," the key length will be required to be `24` bytes.
    - This is actually three times the length of a "`DES`" key, which is required to be `8` bytes.
    - The `String` "`The SuperSecret Password`" is being used since it has 24 characters that will be converted into bytes.

Ciphers

CCEncrypter.java

```
 1. package examples.security;
 2.
 3. import java.security.Key;
 4.
 5. import javax.crypto.Cipher;
 6. import javax.crypto.spec.SecretKeySpec;
 7.
 8. public class CCEncrypter {
 9.     private static final String algorithm = "DESede";
10.     private static Key key;
11.     private static Cipher cipher;
12.     static {
13.         try {
14.             String ssp = "The SuperSecret Password";
15.             key = new SecretKeySpec(ssp.getBytes(),
16.                     algorithm);
17.             cipher = Cipher.getInstance(algorithm);
18.         } catch (Exception e) {
19.             String msg =
20.                 "Unable to initialize CCEncrypter";
21.             System.err.println(msg);
22.             e.printStackTrace();
23.         }
24.     }
25.
26.     public static String decrypt(byte data []) throws
27.             Exception {
28.         cipher.init(Cipher.DECRYPT_MODE, key);
29.         return new String(cipher.doFinal(data));
30.     }
31.
32.     public static byte [] encrypt(String data) throws
33.             Exception {
34.         cipher.init(Cipher.ENCRYPT_MODE, key);
35.         return cipher.doFinal(data.getBytes());
36.     }
37. }
```

- The secret key would normally be stored elsewhere rather than hard coded in the application.
  - Java key stores provide such a mechanism if desired.

Summary

- This chapter introduced various topics in regards to securing Java applications and the resources upon which they rely.
  - We saw some of the APIs and tools provided as part of the JDK to assist a developer with security issues.
    - The bytecode verifier is used by the JVM at runtime to ensure that only legal code is executed at runtime.
    - Access control can be maintained using a security manager, policy files, and permissions.
    - Data can be protected from prying eyes using a cipher to encrypt and/or decrypt the data.
- Additional information on each of the above topics and many additional topics not shown here in this chapter is available as part of the documentation that is available with the JDK.
  - This information can be found in the following location.

    `${JAVA_HOME}/docs/technotes/guides/security/index.html`

    - Where `${JAVA_HOME}` is the directory on your computer where Java was installed.
  - If the information is not available on your local hard drive it can also be referenced from Sun's website at the following URL.

    `http://java.sun.com/javase/technologies/security/`

Exercises

1. Use the `policytool` to grant additional permissions such that when the DetailedResources application is run with a security manager in place, that all of the code still executes without any exceptions being generated.

2. Working with a copy of one the examples or exercises from the Advanced I/O chapter that serializes objects, use the CCEncrypter class to protect one or more pieces of data being persisted.

   - It is important to note that the default padding used in the `CCEncrypter` will result in the final length in bytes of the resultant cipher text will a multiple of 8.

     - For example, a 10 digit phone number of the form `(xxx) xxx-xxxx` which is 14 characters would be padded to a resultant cipher text of length 16 bytes.
     - This will help in knowing how many bytes to read from the stream.

Chapter 10:
Logging API

1) Introduction 10-2

2) Loggers 10-3

3) Logger Levels 10-5

4) Logger Handlers 10-6

5) Specifying Handlers and Formatters 10-9

6) Configuring Handlers 10-10

7) LogManager 10-13

Introduction

- The `java.util.logging` package was introduced in Java SE 1.4 in order to support maintaining and servicing software.

- The key classes and interfaces of this package include the following.
  - `Logger`
    - The main entity on which applications make logging calls
  - `Level`
    - Defines a set of standard logging levels that can be used to control logging output
  - `LogRecord`
    - Used to pass logging requests between the logging framework and individual log handlers
  - `Handler`
    - Exports `LogRecord` objects to a variety of destinations including memory, output streams, consoles, files, and sockets
  - `Formatter`
    - Provides support for formatting `LogRecord` objects. This package includes two formatters (`SimpleFormatter` and `XMLFormatter`) for formatting log records in plain text or XML respectively
  - `Filter`
    - Provides fine-grained control over what gets logged, beyond the control provided by log levels

Loggers

- A `Logger` object is used to log messages for a specific system or application component.
  - A `Logger` object may be obtained by calling one of the static `getLogger` factory methods.
    - These will either create a new `Logger` or return a suitable existing `Logger`.
  - A `Logger` is normally named using a hierarchical dot-separated namespace.
    - `Logger` names should normally be based on the package name or class name of the logged component.
  - Each `Logger` keeps track of a "parent" `Logger`, which is its nearest existing ancestor in the `Logger` namespace.
  - Each `Logger` has a `Level` with which it is associated.
    - This reflects a minimum `Level` that this logger cares about.
    - If a level is set to `null`, then its effective level is inherited from its parent, which may in turn obtain it recursively from its parent, and so on up the tree.
  - Logging messages will be forwarded to registered `Handler` objects, which can forward the messages to a variety of destinations, including consoles, files etc.
- The example on the next page demonstrates how a `Logger` is generated and shows several methods from the `Logger` class that will be used to generate logging messages.

Loggers

LogTest1.java

```
 1.  package examples.logging;
 2.
 3.  import java.util.logging.Logger;
 4.
 5.  public class LogTest1 {
 6.
 7.      private static Logger log =
 8.         Logger.getLogger("examples.logging.LogTest1");
 9.
10.      public static void main(String args[]){
11.          log.finest("Logged at Finest level");
12.          log.finer("Logged at Finer level");
13.          log.fine("Logged at Fine level");
14.          log.config("Logged at Config level");
15.          log.info("Logged at Info level");
16.          log.warning("Logged at Warning level");
17.          log.severe("Logged at Severe level");
18.      }
19.  }
```

- When creating a `Logger`, it is recommended that the fully qualified name of the class, in which the `Logger` is defined, be used, as shown in the example above.
  - The default configuration of a `Logger` sends logging messages to the console (`System.err`).
  - As seen from the output of the above program, the default `Logger` will only log records at or above a certain threshold level (`Level.INFO`).

```
Jan 8, 2007 9:21:21 AM examples.logging.LogTest main
INFO: Info level logged
Jan 8, 2007 9:21:22 AM examples.logging.LogTest main
WARNING: Warning level logged
Jan 8, 2007 9:21:22 AM examples.logging.LogTest main
SEVERE: Severe level logged
```

Logger Levels

- The logging API defines a `Level` class to specify the importance of a given logging record.
  - The `Level` class has a set of predefined logging levels as shown in the table below.

Level	Intended Indication
`SEVERE` (Highest value)	Serious errors such as a fatal program error
`WARNING`	Potential problems
`INFO`	Informational runtime messages
`CONFIG`	Configuration settings
`FINE`	Debugging problems
`FINER`	Greater detailed debugging problems
`FINEST` (Lowest value)	Highly detailed debugging problems

  - In addition to the above levels, there are two special levels.
    - `ALL` - enables logging of all records
    - `OFF` - used to turn logging off

- The `Logger` class has convenience methods for logging each of the levels in the table above, as demonstrated in the example on the previous page.
  - There is also a `log` method in the `Logger` class that takes two parameters, the logging `Level` and a `String` representing the message to be logged.
  - The `setLevel` method of a `Logger` allows the threshold level of logging to be specified for the given `Logger`.

Logger Handlers

- In order to control the threshold `Level` that a `Logger` uses, the `setLogger` method can be called on the `Logger` with the desired `Level`.

- The actual output of the record being logged is delegated to a subclass of `Handler` by the `Logger`.

- Since a `Handler` has its own `Level`, it is also necessary to set the `Level` of the `Handler` in order to completely control the output of a `Logger`.
  - The root logger's handlers default to `Level.INFO`.
    - In order to change the root handler's level, it is first necessary to obtain a reference to the root handler.
    - This can be achieved by passing the empty string to the `getLogger` method.

- The example on the next page expands upon the previous example by including the necessary code to change the threshold logging level of both the `Logger` and the `Logger`'s `Handler`(s).

Logger Handlers

LogTest2.java

```
 1. package examples.logging;
 2.
 3. import java.util.logging.Handler;
 4. import java.util.logging.Level;
 5. import java.util.logging.Logger;
 6.
 7. public class LogTest2 {
 8.
 9.     private static Logger log =
10.        Logger.getLogger("examples.logging.LogTest2");
11.
12.     public static void main(String args[]){
13.
14.         Handler [] handlers =
15.             Logger.getLogger("").getHandlers();
16.
17.         for(int i = 0; i < handlers.length; i++){
18.             handlers[i].setLevel(Level.FINE);
19.         }
20.
21.         log.setLevel(Level.FINE);
22.
23.         log.finest("Logged at Finest level");
24.         log.finer("Logged at Finer level");
25.         log.fine("Logged at Fine level");
26.         log.config("Logged at Config level");
27.         log.info("Logged at Info level");
28.         log.warning("Logged at Warning level");
29.         log.severe("Logged at Severe level");
30.     }
31. }
```

- The example code, in bold above, sets the `Level` of both the class' `Logger` and the root logger's handlers to a level of `Level.FINE`.

Logger Handlers

- The logging API defines the following concrete subclasses of the `Handler` class.
  - `StreamHandler` - Writes formatted records to an `OutputStream`
  - `ConsoleHandler` - Writes formatted records to `System.err`
  - `FileHandler` - Writes formatted records either to a single file, or to a set of rotating log files
  - `SocketHandler` - Writes formatted records to remote TCP ports
  - `MemoryHandler` - Buffers records in memory
- The API also includes two standard Formatters.
  - `SimpleFormatter` - Writes brief "human-readable" summaries of log records
  - `XMLFormatter` - Writes detailed XML-structured information
- The root logger, by default, uses a `ConsoleHandler` that relies upon a `SimpleFormatter`, resulting in summaries being sent to `System.err`.
- The example on the following page demonstrates the use of two `FileHandler` objects.
  - One `FileHandler` will use a `SimpleFormatter`.
  - The other `FileHandler` will use an `XMLFormatter`.
  - Both `FileHandler` objects will be used by a single `Logger`.

Specifying Handlers and Formatters

LogTest3.java

```
 1. package examples.logging;
 2.
 3. import java.io.IOException;
 4. import java.util.logging.*;
 5.
 6. public class LogTest3 {
 7.
 8.     private static Logger log =
 9.        Logger.getLogger("examples.logging.LogTest3");
10.
11.     public static void main(String args[]){
12.         try {
13.             Handler h1 = new FileHandler("log.txt");
14.             Handler h2 = new FileHandler("log.xml");
15.             h1.setFormatter(new SimpleFormatter());
16.             log.addHandler(h1);
17.             h2.setFormatter(new XMLFormatter());
18.             log.addHandler(h2);
19.         }catch (IOException e) {
20.             String msg = "Unable to create log file"
21.                 + ":Using System defaults instead";
22.             log.warning(msg);
23.         }
24.
25.         log.setLevel(Level.FINE);
26.
27.         log.finest("Logged at Finest level");
28.         log.finer("Logged at Finer level");
29.         log.fine("Logged at Fine level");
30.         log.config("Logged at Config level");
31.         log.info("Logged at Info level");
32.         log.warning("Logged at Warning level");
33.         log.severe("Logged at Severe level");
34.     }
35. }
```

- The above program will output the logging information to three different destinations, which are the file `log.txt`, the file `log.xml`, and the console `System.err`.
  - However, `System.err` will only show `Level.INFO` or higher.

Configuring Handlers

- Each subclass of `Handler` defines various constructors and/or methods for configuring the properties of the `Handler`.
  - The notes below detail the configuration of a `FileHandler`.
  - Other `Handler` properties can be found in each subclass' respective documentation.

*Property	Description	Default Value
`level`	Default `Level` for the `Handler`	`Level.All`
`filter`	Name of a `Filter` class to use	no `Filter`
`formatter`	Name of a `Formatter` class to use	`XMLFormatter`
`encoding`	Name of the character set encoding to use	default platform encoding
`limit`	Maximum (in bytes) to write to any one file. If this is zero, then there is no limit	0
`count`	Number of output files to cycle through	`1`
`pattern`	Pattern for generating the output file name	`"%h/java%u.log"`
`append`	Indicates whether the `FileHandler` should append onto any existing files	`false`

  - *Each Property listed in the above left column should be pre-pended with `java.util.logging.FileHandler` (i.e., `java.util.logging.FileHandler`**`.level`**`).`
  - The `java.util.logging.FileHandler.pattern` property is described in more detail on the following page.

Configuring Handlers

- A pattern consists of a string, including the following special components that will be replaced at runtime.

Pattern	Description
`"/"`	The local pathname separator
`"%t"`	The system temporary directory
`"%h"`	The value of the "`user.home`" system property
`"%g"`	The generation number to distinguish rotated logs
`"%u"`	A unique number to resolve conflicts
`"%%"`	Translates to a single percent sign "%"

- The `FileHandler` class defines:
  - constructors to specify the `limit`, `count`, `pattern`, and `append` properties; and
  - set methods to specify the `level`, `filter`, `formatter`, and `encoding` properties.

- The example on the next page defines a `FileHandler` with the following properties.
  - Limits the file size to 2k
  - Cycles through four logs before overwriting oldest first.
  - Appends information to file rather than overwriting
  - Creates the log files in the users' temp directory with the following naming pattern.
    - `sampleLog0.log`
    - `sampleLog1.log`
    - `sampleLog2.log`
    - `sampleLog3.log`

Configuring Handlers

LogTest4.java

```
 1. package examples.logging;
 2. import java.util.logging.*;
 3.
 4. public class LogTest4 {
 5.
 6.     private static Logger log =
 7.        Logger.getLogger("examples.logging.LogTest4");
 8.
 9.     public static void main(String args[]){
10.         String p = "%tsampleLog%g.log";
11.         try {
12.             Handler h =
13.                 new FileHandler(p, 2000, 4, true);
14.             h.setFormatter(new SimpleFormatter());
15.             log.addHandler(h);
16.         }catch (java.io.IOException e) {
17.             String msg = "Unable to create log file"
18.                 + ":Using System defaults instead";
19.             log.warning(msg);
20.         }
21.
22.         // turn off root loggers handlers
23.         Handler [] handlers =
24.             Logger.getLogger("").getHandlers();
25.         for(int i = 0; i < handlers.length; i++){
26.             handlers[i].setLevel(Level.OFF);
27.         }
28.
29.         // set class logger to FINE
30.         log.setLevel(Level.FINE);
31.         int loopCount = 40;
32.         if(args.length == 1)
33.             loopCount = Integer.parseInt(args[0]);
34.         for(int i = 0; i < loopCount; i++){
35.                 log.info("Msg #" + i + " logged");
36.         }
37.         System.out.println("Log File Updated with " +
38.             loopCount + " new records");
39.     }
40. }
```

LogManager

- `Handler` classes typically use `LogManager` properties to set default values for the `Handler`.
  - Each concrete `Handler` class defines its own set of properties.
  - This allows each Logger to be configured and changed through properties files rather than in the business logic.
- There is a single `LogManager` object used to maintain a set of shared state about `Logger`s and log services.
  - The `LogManager` object manages a:
    - hierarchical namespace of `Logger` objects that stores all named `Logger`s; and
    - set of logging control properties, consisting of simple key-value pairs that can be used by `Handler`s and other logging objects.
  - The `LogManager` object is created during class initialization at startup.
    - A reference to the `LogManager` object can be retrieved using `LogManager.getLogManager()`.
- By default, the `LogManager` reads its initial configuration from a properties file "lib/logging.properties" in the JRE directory.
  - If you edit that property file, you can change the default logging configuration for all uses of that JRE.

LogManager

- The `LogManager` also permits the use of two optional system properties that allow more control over reading the initial configuration.
  - `java.util.logging.config.class`
  - `java.util.logging.config.file`
    - These two properties may be set via the `Preferences` API, or as command line property definitions to the "java" command.
- The properties file shown below is designed to mirror the properties configured in the previous example.
  - Note that some of the properties in the code below take up two lines in order to fit them on the printed page, whereas the actual properties file requires that each be completely defined on one line.

sampleConfig.properties

```
1.  ############################################################
2.  #   Global properties of root Logger
3.  ############################################################
4.  handlers = java.util.logging.ConsoleHandler,
5.       java.util.logging.FileHandler
6.  .level = OFF
7.  java.util.logging.FileHandler.pattern = %h/java%u.log
8.  java.util.logging.FileHandler.limit = 50000
9.  java.util.logging.FileHandler.count = 1
10. java.util.logging.FileHandler.formatter =
11.      java.util.logging.XMLFormatter
12. java.util.logging.ConsoleHandler.level = INFO
13. java.util.logging.ConsoleHandler.formatter =
14.      java.util.logging.SimpleFormatter
15.
```

LogManager

***sampleConfig.properties* continued**

```
16. ####################################################
17. # Specific properties.
18. # Provides extra control for each logger.
19. ####################################################
20.
21. examples.logging.LogTest5.handlers =
22.      examples.logging.SampleFileHandler
23. examples.logging.LogTest5.level = FINEST
24. examples.logging.SampleFileHandler.pattern =
25.      %t/sampleLog%g.log
26. examples.logging.SampleFileHandler.limit = 2000
27. examples.logging.SampleFileHandler.count = 4
28. examples.logging.SampleFileHandler.formatter =
29.      java.util.logging.SimpleFormatter
30. examples.logging.SampleFileHandler.level = FINEST
```

- A subclass of `FileHandler`, named `SampleFileHandler.java`, was created in order to control logging to a file separately from the `FileHandler` potentially used by the root logger.
  - The `SampleFileHandler` class relies entirely upon the behavior of its parent class by simply echoing the available constructors of `FileHandler`.
- The example on the next page is a simplification of the previous example in that no properties of the logger or its handler(s) are specified in the code itself.

LogManager

LogTest5.java

```
 1. package examples.logging;
 2. import java.util.logging.*;
 3.
 4. public class LogTest5 {
 5.
 6.     private static Logger log =
 7.        Logger.getLogger("examples.logging.LogTest5");
 8.
 9.     public static void main(String args[]){
10.         int loopCount = 40;
11.         if(args.length == 1)
12.             loopCount = Integer.parseInt(args[0]);
13.         for(int i = 0; i < loopCount; i++){
14.             log.info("Msg #" + i + " logged");
15.             log.finest("Msg #" + i + " logged");
16.         }
17.         System.out.println("Log File Updated with " +
18.             loopCount + " new records");
19.     }
20. }
```

- In order to have the `LogManager` read from the `sampleConfig.properties` file when the `LogTest5` program above is run, the following can be typed on the command line.

  ```
  java -Dk=v examples.logging.LogTest5
  ```

  - The "**k**" above should be replaced with:

    ```
    java.util.logging.config.file
    ```

  - The "**v**" above should be replaced with:

    ```
    C:\AdvancedJavaProject\src\examples\logging\sampleConf
    ig.properties
    ```

Exercises

1. Add the following logging capabilities to the `ProductDBFromFile` class created in an earlier exercise.
   - Each time a `Product` is placed in `ProductDBFromFile` object successfully, it should be logged at the `FINE` level.
   - If the `open` method of the class fails (returns a `boolean` of `false`), it should be logged at the `WARNING` level.
2. Add the following logging capabilities to the `Configuration` class created earlier.
   - Each key/value pair set as a `System` property should be logged at the `CONFIG` level.
   - If the `loadConfiguration` method of the class fails (returns a `boolean` of `false`), it should be logged at the `WARNING` level.

This Page Intentionally Left Blank

Chapter 11:
Networking

1) Networking Fundamentals **11-2**

2) The Client/Server Model **11-4**

3) `InetAddress` **11-6**

4) URLs **11-8**

5) Sockets **11-11**

6) A Time-of-Day Client **11-13**

7) Writing Servers **11-14**

8) Client/Server Example **11-15**

Networking Fundamentals

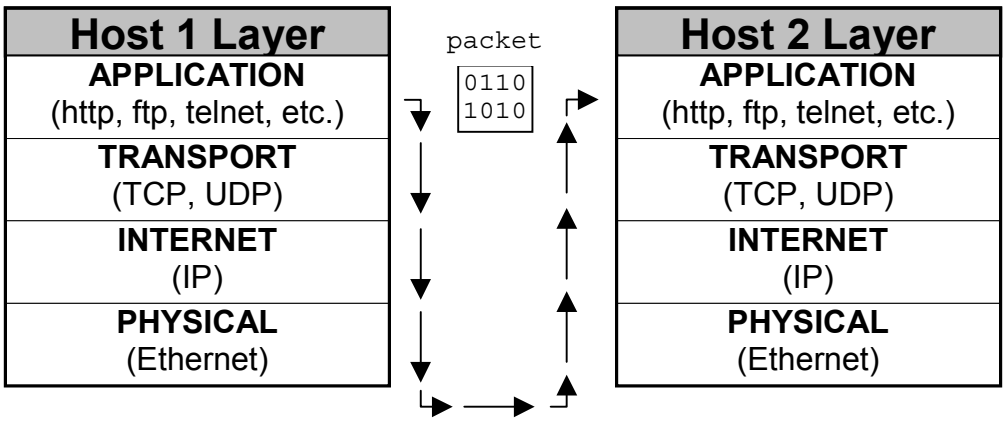
- Before looking at the Java networking examples, we will give some basic details of networking in general.
  - Every machine on a network is called a node or a **host**. Each host has an **address**, which uniquely identifies it on the network.
    - The address is a four-byte number, usually represented as four numbers separated by dots.

      ```
      127.0.0.1     192.168.0.100
      ```

    - Since humans are better at remembering names than numbers, a system has been developed to map names to network addresses. The **D**omain **N**ame **S**ystem (**DNS**) is incorporated into machines known as **Name Servers**.
  - Data sent over a network is sent in **packets**. Each packet contains the address of the sender and receiver.
  - Computers communicate through a set of rules called **protocols**.
    - Although there are many protocols in use, we are concerned with only the **http** and **tcp/ip** protocols.
  - When a process on one machine wishes to send a packet of information to a process on another machine, there are several layers through which the packet must travel.
    - These layers have been described by several models, most notably the seven layer model, known as the **ISO/OSI** model.
    - Since we are interested in providing the networking details for Java programs, we will use a simplified version of the seven layer model.

Networking Fundamentals

- When a packet flows from **Host 1** to **Host 2**, the packet is sent through the layers as depicted in the diagram below.



- Each layer has distinct duties to perform.
  - The application layer is usually the only one with which a Java programmer is concerned.
    - The application layer sends data from or delivers data to your program.
  - The transport layer controls how the packets are delivered. The two most common protocols for this layer are described below.
    - **T**ransmisson **C**ontrol **P**rotocol (**TCP**) - reiliable, high overhead
    - **U**ser **D**atagram **P**rotocol (**UDP**) - unreliable, low overhead
- Each computer on a network can provide many networking services.
  - For example, your computer may provide a **F**ile**T**ransfer **P**rotocol (**FTP**) service and a **H**yper**T**ext **T**ransfer **P**rotocol (**HTTP**) service.

The Client/Server Model

- When an application on one machine needs a particular service on another machine, it is not enough for the application to reference the other machine by address alone.
  - It must reference the service on the machine it wishes to use.
  - Services are referenced by **port** numbers. Each service is mapped to a port. Standardized services use standard ports. For example:
    - the standard ftp port is 21;
    - the standard http is port 80; and
    - the standard telnet is port 23.
- Most networking applications today use the **Client/Server** model.
  - In this model, a process on one machine requests some service, such as a file service or time-of-day service from another process.
    - The process requesting the service is called the **client process**.
    - The process providing the service is called the **server process**.
  - The two processes could reside on the same machine, but more often, they reside on different machines.
    - The machine hosting the client process is called the **Client**.
    - The machine hosting the server process is called the **Server**.

The Client/Server Model

- One example of the Client/Server model is the Browswer/Web Server model.
  - Data is stored on a Server that services many requests from Client software hosted by PCs.
- Browsers and web servers communicate using the HTTP protocol.
  - The HTTP protocol defines how a client requests data from a server, and the data is then transferred back to the client.
    - A browser typically makes a request to the server for a file to retrieve.
    - The name of the file and its location on the World Wide Web is specified as a **U**niform **R**esource **L**ocator (**URL**).
- A URL is a reference to a resource on the Internet. Most people are familiar with URLs as Domain Names such as:
  - `www.trainingetc.com`;
  - `www.sun.com`; or
  - `www.apache.org`.
- In reality, a domain name is just one part of a URL.
  - A URL can consist of the following parts, some of which are dependent upon the protocol.
    - protocol:// `http://`
    - hostname:port `www.trainingetc.com:80`
    - path `courses/file#java`
    - file `courses/file`
    - #section `#java`

InetAddress

- The `java.net` package contains the majority of the classes that pertain to networking in Java.
  - The first class we will study in this package is the `InetAddress` class.
- `InetAddress` can be used to obtain information about a host name or an IP address.
  - The JVM will rely on the DNS server configured for the system to obtain the information.
  - The example on the next page calls various methods from the `InetAddress` class to display information about both the host name provided on the command line and the local host.

InetAddress

Address.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. public class Address {
 4.     public static void main(String args[])  {
 5.         try {
 6.             print("Remote Host:");
 7.             InetAddress remote =
 8.                 InetAddress.getByName(args[0]);
 9.             getInfo(remote);
10.             print("Local Host:");
11.             InetAddress local =
12.                 InetAddress.getLocalHost();
13.             getInfo(local);
14.         } catch(UnknownHostException e) {
15.             print("UnknownHost: " + e.getMessage());
16.         }
17.     }
18.     private static void getInfo(InetAddress ia){
19.         print("  HostName: " +
20.               ia.getHostName());
21.         print("  HostAddress: "
22.               + ia.getHostAddress());
23.         print("  CanonicalHostname: " +
24.               ia.getCanonicalHostName());
25.         getRawIP(ia);
26.         System.out.println();
27.     }
28.     private static void getRawIP(InetAddress ia){
29.         byte [] b = ia.getAddress();
30.         System.out.print("  ");
31.         for (int i = 0; i < b.length; i++) {
32.             int each = b[i] < 0 ? b[i] + 256 : b[i];
33.             System.out.print(each + " ");
34.         }
35.         System.out.println();
36.     }
37.     private static void print(String s){
38.         System.out.println(s);
39.     }
40. }
```

URLs

- `URL` is another class in the `java.net` package.
  - A `URL` can be constructed in various ways, several of which are shown below.

```
public URL(String spec) throws MalformedURLException

public URL(String protocol, String host,
           String file) throws MalformedURLException

public URL(String protocol, String host, int port,
           String file) throws MalformedURLException
```

  - Once a `URL` is constructed, several methods can be used to retrieve a specific field, as shown in the example below.
    - The application below requires a host name be supplied on the command line.

URLS.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. public class URLS {
 4.     public static void main(String args[]) {
 5.         try {
 6.             URL u = new URL("http", args[0], 80,
 7.                             "/index.html");
 8.             System.out.println(u);
 9.             print("Prot: " + u.getProtocol());
10.             print("Host: " + u.getHost());
11.             print("Port: " + u.getPort());
12.             print("Ref:  " + u.getRef());
13.             print("File: " + u.getFile());
14.         } catch(MalformedURLException e) {
15.             System.out.println(e.getMessage());
16.         }
17.     }
18.     private static void print(String s){
19.         System.out.println(s);
20.     }
21. }
```

URLs

- The `openStream` method in the `URL` class returns an `InputStream` that can be used to read the data at the given `URL` as shown in the example below.

ReadWebPage.java

```
 1.  package examples.networking;
 2.  import java.net.*;
 3.  import java.io.*;
 4.  public class ReadWebPage {
 5.      public static void main(String args[]) {
 6.          try {
 7.              URL web = new URL("http://" + args[0]);
 8.              InputStream  is = web.openStream();
 9.              BufferedReader br = new BufferedReader(
10.                      new InputStreamReader(is));
11.              String text;
12.              while((text = br.readLine()) != null)
13.                  System.out.println(text);
14.              br.close();
15.          } catch(MalformedURLException e)  {
16.              System.out.println("Malformed");
17.          } catch(IOException e) {
18.              System.out.println("IOException");
19.          }
20.      }
21.  }
```

- Below is some sample output from the above program.

```
java examples.networking.ReadWebPage
www.trainingetc.com
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<title>/training/etc Inc. Home Page</title>
<link href="/css/navskip.css" rel="stylesheet"
type="text/css"/>
<link href="/css/main.css" rel="stylesheet"
type="text/css"/>
```

URLs

- A `URLConnection` can be obtained from the `openConnection` method of the `URL` class.
  - The `URLConnection` class uses various methods to obtain the header information sent by the server instead of the body of the response from the server as in the previous example.
    - Certain header fields that tend to be accessed frequently have special methods to obtain them.
  - The example below demonstrates using the `URLConnection` to obtain information about the headers sent by the server.

ReadHeaders.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. import java.io.*;
 4. import java.util.*;
 5. public class ReadHeaders {
 6.     public static void main(String argv[]) {
 7.         try {
 8.             URL u = new URL("http://" + argv[0]);
 9.             URLConnection uc = u.openConnection();
10.             print("Content Type:" +
11.                 uc.getContentType());
12.             print("Content Length:" +
13.                 uc.getContentLength());
14.             print("Date:" + new Date(uc.getDate()));
15.             print("Last Modified:" +
16.                 new Date(uc.getLastModified()));
17.         } catch(MalformedURLException e) {
18.             System.out.println("Malformed");
19.         } catch(IOException e) {
20.             System.out.println("IOException");
21.         }
22.     }
23.     private static void print(String s){
24.         System.out.println(s);
25.     }
26. }
```

Sockets

- Now we will show a few examples of the communication between Clients and Servers.
  - This typically involves the use of the `Socket` and `ServerSocket` classes.
- A `Socket` is an endpoint of a link between two programs running on a network.
  - A `Socket` uses a port number to identify the application that the transport layer should send the data to as it arrives, or is delivered, over the internet.
    - The `Socket` class is used by Clients and Servers to communicate over the network.
  - A typical Client application needs to:
    - connect to a remote machine;
    - send and/or receive data over the connection; and
    - close the connection.
  - A typical Server application needs to:
    - bind to a port;
    - listen for a connection;
    - accept connections on the port;
    - send and/or receive data over the connection; and
    - close the connection.
- The `Socket` class has several constructors, most of which take an address and a port for the first two parameters.

Sockets

- The example below is a simple Client that attempts to connect to a Server listening for connections on port 80.
  - The constructor for the `Socket` will need to obtain the host name to connect from the command line.
  - The application simply demonstrates several of the methods available in the `Socket` class that allow a developer to obtain information about the `Socket`.

SocketInfo.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. import java.io.*;
 4. public class SocketInfo {
 5.     public static void main(String args[]) {
 6.         try {
 7.             Socket s = new Socket(args[0], 80);
 8.             print("Connected:")
 9.             print("To: " + s.getInetAddress());
10.             print("  on port  " + s.getPort());
11.             print("From " + s.getLocalAddress());
12.             print("  on port " + s.getLocalPort());
13.             s.close();
14.         } catch(UnknownHostException e) {
15.             print("Unknown Host: " + args[0]);
16.             e.printStackTrace();
17.         } catch(SocketException e) {
18.             print("SocketException: " + args[0]);
19.             e.printStackTrace();
20.         } catch(IOException e) {
21.             print("IOException:");
22.             System.out.println(e);
23.             e.printStackTrace();
24.         }
25.     }
26.     private static void print(String s){
27.         System.out.println(s);
28.     }
29. }
```

A Time-of-Day Client

- Now we will show an example of a client that connects to a Daytime service (a service supplied by many Servers on port 13).
  - In order to do this, we need to know how to read and write with sockets.
  - The `Socket` class has a pair of methods that will allow communication.

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

  - In the example below, data will be flowing in only one direction (from the Server to the Client). Therefore, we only need to rely on the `getInputStream` method of the `Socket`.

DayTimeClient.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. import java.io.*;
 4. public class DayTimeClient {
 5.     public static void main(String args[]) {
 6.         String host = "time.nist.gov";
 7.         try {
 8.             if(args.length > 0)
 9.                 host = args[0];
10.             Socket s = new Socket(host, 13);
11.             InputStream is = s.getInputStream();
12.             System.out.println("Time at " +
13.                                host + " is");
14.             int data;
15.             while((data = is.read()) != -1)
16.                 System.out.print((char) data);
17.             is.close();
18.             s.close();
19.         }catch(IOException e) {
20.             System.out.println(e);
21.         }
22.     }
23. }
```

Writing Servers

- Before we look at a Java Client application communicating with a Java Server application, we need to know a little bit more about how Servers are written in Java.

- The `ServerSocket` class is similar to the `Socket` class except that it provides methods for the additional tasks for which a server is typically responsible.

- A Server typically has the following life cycle.
  - Bind to a particular port during its construction.
  - Listen for a Client on that port by calling `accept()` from the `ServerSocket` class.
  - Use the `Socket` returned from the `accept` method to then call the `getInputStream` and/or `getOutputStream` method(s) on the `Socket`.
  - The business of the Client and the Server is then transacted.
  - The connection is closed by either the Client or the Server.
  - Typically, the Server then waits for another connection.

- Several forms of the `ServerSocket` constructor are shown below.

```
public ServerSocket(int port) throws IOException
```

  - Binds the Server to the specified port

```
public ServerSocket(int port, int backlog) throws
IOException
```

  - Binds the Server to the specified port, and sets the maximum queue length for incoming, pending connections

Client/Server Example

- The following example demonstrates an "Echo Server," which echoes whatever data it receives back to the client in upper case.
  - Comments have been placed in the code to indicate the various aspects of the life-cycle of the Server.

EchoServer.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. import java.io.*;
 4. public class EchoServer {
 5.     public static void main(String args[]) {
 6.         ServerSocket theServer = null;
 7.         Socket clientSocket;
 8.         int port = 2345;
 9.         InetAddress ia = null;
10.         // Attempt to start the server
11.         // bound to the given port
12.         try{
13.             theServer = new ServerSocket(port);
14.             // Print info about the server
15.             ia = InetAddress.getLocalHost();
16.             String host = ia.getHostAddress();
17.             System.out.print("Server started on " +
18.                 host+ "  Listening on port "+ port);
19.             // loop for each client
20.             while(true){
21.                 // wait for a client to connect
22.                 clientSocket = theServer.accept();
23.                 // handle client in a helper method
24.                 handleClient(clientSocket);
25.             } // proceed to next Client
26.         } catch(IOException ioe){
27.             ioe.printStackTrace();
28.             System.exit(1);
29.         }
30.     }
```

Code continued on following page

Client/Server Example

EchoServer.java - continued

```
31.      // Helper method to handle client communications
32.      private static void handleClient(Socket cSocket){
33.          PrintStream toClient;
34.          BufferedReader fromClient;
35.          String data;
36.          try{
37.              // Get Input and Output
38.              fromClient = new BufferedReader(
39.                  new InputStreamReader(
40.                      cSocket.getInputStream()));
41.              toClient = new PrintStream(
42.                  cSocket.getOutputStream());
43.              while(true){
44.                  // read from Client
45.                  data = fromClient.readLine();
46.                  if(data == null) break;
47.                  data = data.toUpperCase();
48.                  // write to Client
49.                  toClient.println(data);
50.              }
51.              fromClient.close();
52.              toClient.close();
53.              cSocket.close();
54.          }catch(IOException ioe){
55.              String msg = "Connection lost";
56.              System.out.println(msg);
57.          }
58.      }
59.  }
```

- A Client application, capable of communicating with the above Server, is shown on the next page.

Client/Server Example

EchoClient.java

```
 1. package examples.networking;
 2. import java.net.*;
 3. import java.io.*;
 4. public class EchoClient {
 5.     public static void main(String args[]) {
 6.         Socket con = null;
 7.         PrintStream toServer;
 8.         BufferedReader fromServer, fromKB;
 9.         String data;
10.         int port = 2345;
11.         String host = "localhost";
12.         if(args.length > 0)
13.             host = args[0];
14.         try{
15.             // Attempt to connect to server
16.             con = new Socket(host, port);
17.         } catch(IOException ioe){
18.             // No use in continuing
19.             String msg = "Unable to connect";
20.             System.out.println(msg);
21.             ioe.printStackTrace();
22.             System.exit(1);
23.         }
24.         try{
25.             // get Input(s) and Output
26.             fromKB = new BufferedReader(
27.                 new InputStreamReader(
28.                     System.in));
29.             fromServer = new BufferedReader(
30.                 new InputStreamReader(
31.                     con.getInputStream()));
32.             toServer = new PrintStream(
33.                 con.getOutputStream());
34.             // communicate with the server
35.             String prompt = "Enter Data:\n" +
36.                 "Entering just the word QUIT will " +
37.                 "Close the connection.";
38.             System.out.println(prompt);
```

▸ Code continued on following page

Client/Server Example

EchoClient.java - continued

```
39.             while(true){
40.                 // read from keyboard
41.                 data = fromKB.readLine();
42.                 if(data.equals("QUIT")) break;
43.                 // write data to server
44.                 toServer.println(data);
45.                 // read response from server
46.                 System.out.println(
47.                     fromServer.readLine());
48.             }
49.             // close resources
50.             fromServer.close();
51.             toServer.close();
52.             con.close();
53.         }catch(IOException ioe){
54.             String msg = "Connection lost";
55.             System.out.println(msg);
56.         }
57.     }
58. }
```

- To test the above application, open up a separate DOS window for the `EchoServer` and one DOS window for each `EchoClient`.
  - The server, as written, is only able to handle one client at a time.
    - This is because the while loop that the server uses does not advance to the next iteration until the `handleClient` method returns, which enables the server to wait for another client by calling `accept` again.
    - The work currently performed by the `handleClient` method is a perfect candidate for a thread.

Exercises

1. The `DayTimeClient` application from this chapter relied on the availability of a pre-existing Daytime service.
   - Write your own version of a `DayTimeServer` that is capable of handling the current `DayTimeClient`.
2. Working with copies of both the `DayTimeServer` and `DayTimeClient`:
   - modify the `DayTimeServer` so that it writes a `Date` object to the client (using an `ObjectOutputStream`) rather than the date as a `String`; and
   - modify the `DayTimeClient` to read a `Date` object from the server (using an `ObjectInputStream`) rather than the date as a `String`.

This Page Intentionally Left Blank

Chapter 12:
Threads and Concurrency

1) Review of Fundamentals 12-2

2) Creating Threads by Extending `Thread` 12-3

3) Creating Threads by Implementing `Runnable` 12-4

4) Advantages of Using Threads 12-5

5) Daemon Threads 12-8

6) Thread States 12-10

7) Thread Problems 12-14

8) Synchronization 12-16

9) Performance Issues 12-19

Review of Fundamentals

- To learn about threads, one has to retreat to some fundamental computer terminology.
  - When a program is executing, each of the instructions that compose the program is fetched in sequence from memory into the CPU for processing.
    - The program in execution is called a **process**.
  - It is usually said that a modern computer can execute many processes concurrently. This is called **multi-processing**.
    - In reality, this means that many processes are in various states of execution in any single instant.
    - However, when a computer has a single CPU, only a single instruction may be executed in any given instant. Therefore, true multi-processing may only be achieved when a computer has more than one processor.
    - On a single CPU computer, the fact that many processes are in various states of execution at a single instant is known as **multi-programming**. Each process has its own variables.
  - The ability of a single process to spawn multiple execution paths is called **multi-threading**. Each path is called a **thread**. A thread is also referred to as a lightweight process.
    - Unlike a process, each thread shares the same set of data (variables). If two threads access this data at the same time, there can be synchronization problems.
    - Generally, multi-threading is advantageous, since it allows the same program to handle multiple events "concurrently."
  - There are two ways to create a new thread of execution.
    - Declare a class that extends `Thread`.
    - Declare a class that implements the `Runnable` interface.

Creating Threads by Extending `Thread`

- The `java.lang.Thread` class implements the `Runnable` interface.
  - Therefore, one way of creating a thread is to create a class that extends `Thread` and override the `run` method.
- When a `Thread` is created, it is in the new state.
  - It does not enter the runnable state until it is started.
  - A `Thread` is started by calling its `start` method.
    - This notifies the thread scheduler that a new thread can now be started at some time in the near future.

MyThread.java

```
 1.  package examples.threads;
 2.  public class MyThread extends Thread {
 3.      public MyThread(String s) {
 4.          super(s);
 5.      }
 6.      public void run() {
 7.          for(int i = 0; i < 5; i++){
 8.              System.out.println(getName() + " " + i);
 9.          }
10.      }
11.      public static void main(String a[]) {
12.          MyThread t;
13.          t = new MyThread("Thread A");
14.          t.start();
15.          t = new MyThread("Thread B");
16.          t.start();
17.          for (int i = 0; i < 5; i++)
18.            System.out.println("MainThread " + i);
19.      }
20.  }
```

- There are three threads in the code above, the main thread and the two `MyThread` threads.

Creating Threads by Implementing `Runnable`

- The `Runnable` interface defines a single method named `run`, as shown below.

```
public interface Runnable {
    public void run();
}
```

  - When you implement the `Runnable` interface, you still must create a `Thread` object by passing a reference to your `Runnable` object to the `Thread` objects constructor.
  - The process is demonstrated in the code below.

MyRunnable.java

```
 1. package examples.threads;
 2. public class MyRunnable implements Runnable {
 3.     public MyRunnable() { }
 4.     public void run() {
 5.         String name =
 6.             Thread.currentThread().getName();
 7.         for (int i = 0; i < 5; i++)
 8.             System.out.println(name + " " + i);
 9.     }
10.     public static void main(String args[]) {
11.         Thread t;
12.         t = new Thread(new MyRunnable());
13.         t.start();
14.         t = new Thread(new MyRunnable());
15.         t.start();
16.         String name =
17.             Thread.currentThread().getName();
18.         for (int i = 0; i < 5; i++)
19.             System.out.println(name + " " + i);
20.     }
21. }
```

  - When the thread scheduler decides to run each `Thread` object, it does so by calling the `run` method from the `Runnable` object passed to the `Thread` constructor.

Advantages of Using Threads

- There are several reasons to use threads.
  - Threads can isolate tasks and make programs easier to follow.
  - Threads can make your program run faster.
  - Threads can be used as progress indicators of another thread running in the background.
- In order to get a better understanding of how threads are used and the advantages they offer, we will demonstrate several versions of an application that rely upon the `copy` method in the class shown below.

FileCopyUtility.java

```
 1.  package examples.threads;
 2.  import java.io.*;
 3.  public class FileCopyUtility{
 4.      public static void copy(File src, File dest){
 5.          if (!src.isDirectory()){
 6.              FileInputStream fis = null;
 7.              FileOutputStream fos = null;
 8.              try{
 9.                  fis = new FileInputStream(src);
10.                  fos = new FileOutputStream(dest);
11.                  int theByte;
12.                  while( (theByte = fis.read()) != -1){
13.                      fos.write(theByte);
14.                      // Simulate large file being read
15.                      try{Thread.sleep(10);}
16.                      catch(InterruptedException ie){}
17.                  }
18.                  fis.close();
19.                  fos.close();
20.              } catch(IOException ioe){
21.                  ioe.printStackTrace();
22.              }
23.          }
24.      }
25.  }
```

Advantages of Using Threads

- The application below copies a list of files, whose names are supplied on the command line.
  - This version of the application does not use threads.
    - During each copy process, there is no onscreen indication of the progress, which may result in a user wondering if anything is actually happening.

FileCopier1.java

```
 1.  package examples.threads;
 2.  import java.io.*;
 3.  public class FileCopier1 {
 4.      public static void main(String args[]){
 5.          File temp = new File("C:/output");
 6.          temp.mkdir();
 7.          for(int i = 0; i < args.length; i++){
 8.              File source = new File(args[i]);
 9.              File dest =
10.                  new File (temp, args[i]);
11.              System.out.println();
12.              System.out.println("Copying " + args[i]);
13.              FileCopyUtility.copy(source, dest);
14.          }
15.      }
16.  }
```

- The example on the next page uses a thread as a progress indicator to provide more feedback to the user during the copying process.

Advantages of Using Threads

FileCopier2.java

```
 1. package examples.threads;
 2. import java.io.*;
 3. public class FileCopier2 {
 4.     public static void main(String args[]) {
 5.         File temp = new File("C:/output");
 6.         temp.mkdir();
 7.         Thread t = new ProgressIndicator();
 8.         t.start();
 9.         for(int i = 0; i < args.length; i++){
10.             File source = new File(args[i]);
11.             File dest =
12.                 new File (temp, args[i]);
13.             System.out.println();
14.             System.out.println("Copying " + args[i]);
15.             FileCopyUtility.copy(source, dest);
16.         }
17.         t.interrupt();
18.     }
19. }
```

ProgressIndicator.java

```
 1. package examples.threads;
 2. public class ProgressIndicator extends Thread{
 3.     public void run() {
 4.         while(true){
 5.             System.out.print('.');
 6.             try{
 7.                 Thread.sleep(1000);
 8.             }catch(InterruptedException ie){
 9.                 break;
10.             }
11.         }
12.     }
13. }
```

- Calling the `interrupt` method of the thread results in the thread breaking out of the loop, terminating the application.

Daemon Threads

- Threads can be categorized as either "**user**" or "**daemon**" threads.

- The JVM will continue to run as long as the thread scheduler has at least one "user" thread running.
  - The `main` method of an application runs in a user thread, and if no additional threads are created by the program, the JVM terminates when the end of the `main` method is reached.
    - The application on the previous page created an additional user thread that looped forever, unless the thread was interrupted.

- A daemon thread is normally a low priority thread that runs in the background.
  - The JVM will terminate if the only running threads are daemon threads.
    - The garbage collector is an example of a daemon thread.
  - The example on the following page calls the `setDaemon` method on the thread prior to calling its `start` method.
    - This means that once the `main` method has completed, the only remaining thread will be a daemon thread.
    - Therefore, the JVM will terminate without our code needing to interrupt the thread, as in the previous example.

Daemon Threads

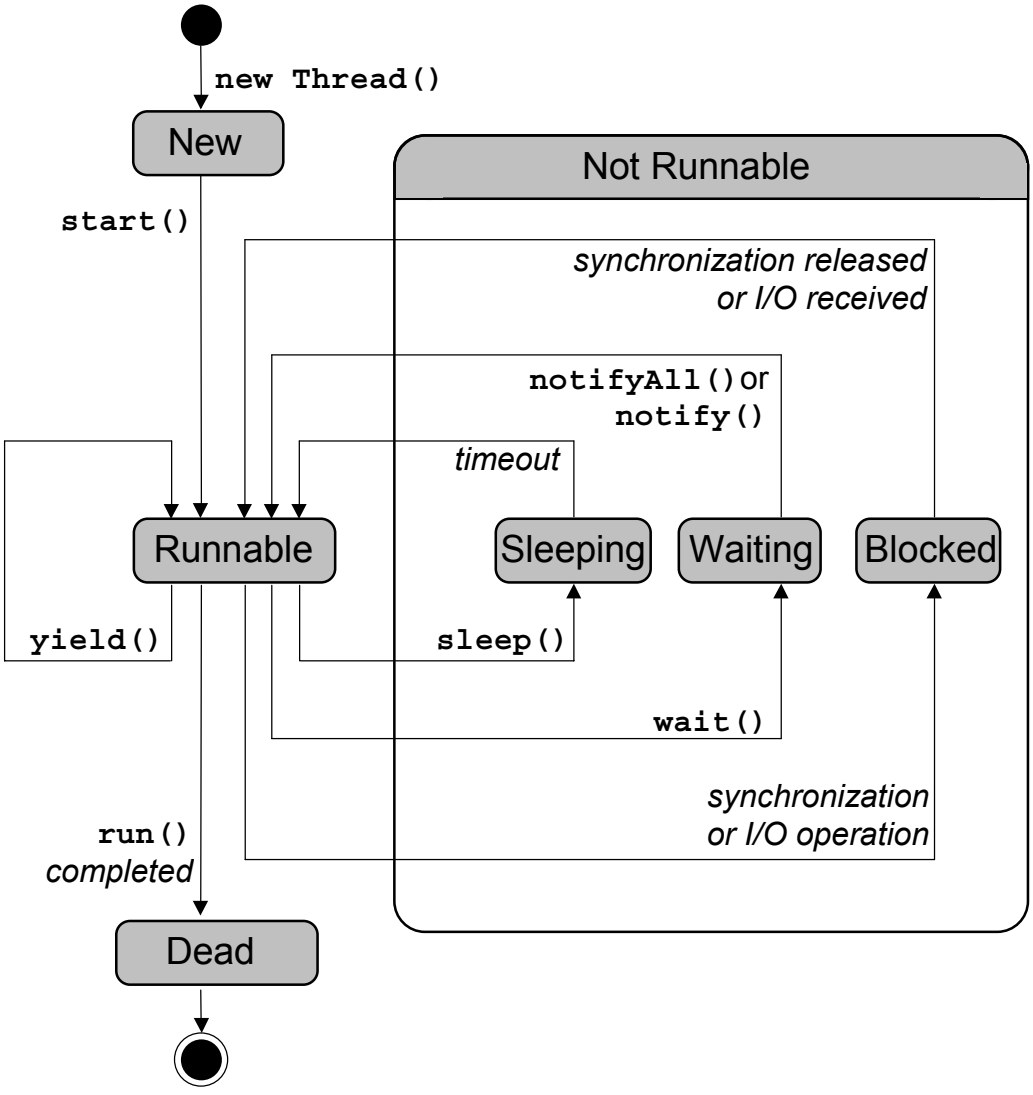
FileCopier3.java

```
 1. package examples.threads;
 2. import java.io.*;
 3. public class FileCopier3 {
 4.     public static void main(String args[]) {
 5.         File temp = new File("C:/output");
 6.         temp.mkdir();
 7.         Thread t = new ProgressIndicator();
 8.         t.setDaemon(true);
 9.         t.start();
10.         for(int i = 0; i < args.length; i++){
11.             File source = new File(args[i]);
12.             File dest =
13.                 new File (temp, args[i]);
14.             System.out.println();
15.             System.out.println("Copying " + args[i]);
16.             FileCopyUtility.copy(source, dest);
17.         }
18.     }
19. }
```

- When the `for` loop above is completed, the `main` method (which runs in a user thread) will have completed.
  - Since the only remaining thread (`t`) was started up as a daemon thread by calling the `setDaemon` method, the JVM will terminate even though the thread (`t`) has not finished.

Thread States

- At any instant, a thread can be in one of various states during its lifetime.
  - The possible thread states are shown in the diagram below.
    - **new** - if the constructor has been called
    - **runnable** - if its `start` method has been called
    - **not runnable** - (blocked) if it's `sleep` or `wait` method has been called, or it is blocking on I/O or `synchronized` code
    - **dead** - if it's `run` method has completed



Thread States

- Some `Thread` methods can be executed only in certain states. An `IllegalThreadStateException` will be thrown otherwise.
  - An example of this would be trying to call that start method on a thread that is in the dead state.
    - This implies that a thread can only be run once.
- The `join` method will cause a thread to wait for the thread the method is called on to die.
- The `yield` method will cause a thread to yield its time to another thread of the same priority.
- The example on the next page will demonstrate some of the methods that control the state of a `Thread`.
  - The application asks the user for keyboard input.
  - If the user does not respond within a certain amount of time, the program will terminate.
    - The timing will be handled by a thread.

Thread States

TimerThread.java

```
 1. package examples.threads;
 2. public class TimerThread implements Runnable {
 3.     int secs;
 4.     public TimerThread(int s) {
 5.             secs = s;
 6.     }
 7.     public void run() {
 8.         System.out.print("Timer set for ");
 9.         System.out.println(secs + " seconds.");
10.         try {
11.             Thread.sleep(secs * 1000);
12.         }
13.         catch(InterruptedException e) {
14.             System.out.print("Timer Thread ");
15.             System.out.println("Interrupted");
16.             return;
17.         }
18.         System.out.println("You are too slow ...");
19.         System.exit(0);
20.     }
21. }
```

- The `TimerThread` above is designed to sleep for a certain number of seconds and terminate the program when it is done sleeping.
  - The only way this thread will not terminate the program it is running within is if the thread gets interrupted.

- The application on the next page contains a loop that:
  - creates an instance of the `Runnable` class defined above;
  - passes the reference to a `Thread` constructor and starts the `Thread`; and
  - requests user input from the keyboard.

Thread States

TimedDataEntry.java

```
 1. package examples.threads;
 2. import java.io.*;
 3. public class TimedDataEntry {
 4.     public static void main(String args[])
 5.       throws IOException{
 6.         String str;
 7.         BufferedReader br = new BufferedReader(
 8.             new InputStreamReader(System.in));
 9.         TimerThread timer;
10.         Thread t;
11.         while(true) {
12.             timer = new TimerThread(10);
13.             t = new Thread(timer);
14.             t.start();
15.             System.out.println("Enter a string: ");
16.             str = br.readLine();
17.             t.interrupt();
18.             System.out.println("You entered " + str);
19.         }
20.     }
21. }
```

- Each time a line of text is successfully read from the keyboard, the `interrupt` method is called to prevent the timer thread from terminating the application.

Thread Problems

- The example on the next page demonstrates some of the problems that can occur when using threads in an application.
  - The application loops through several `int` arrays that are stored in a two dimensional array.
  - Each loop creates a thread to handle the sorting and printing of the contents of one of the `int` arrays.
    - The intended output is to see each array output in sorted order, although which order each array appears in the output does not matter.
  - The static `print` method in the class below will be used to do the actual sorting and printing.

PrintingUtils.java

```
 1. package examples.threads;
 2. import java.util.*;
 3. public class PrintingUtils{
 4.     public static void print(int x[]){
 5.         Arrays.sort(x);
 6.         for(int i = 0; i < x.length; i++){
 7.             System.out.print(x[i]);
 8.             if(i < x.length - 1)
 9.                 System.out.print(",");
10.         }
11.         System.out.println();
12.     }
13. }
```

Thread Problems

- In the example below, each `Thread` (`SyncProblems`) is composed of an `int` array that will be passed to the `PrintingUtils.print` method.

SyncProblems.java

```
 1. package examples.threads;
 2. public class SyncProblems extends Thread{
 3.     public static void main(String args[]){
 4.         int odds [][] = {{9, 8, 7, 6, 5, 4, 3, 2, 1},
 5.                  {52, 22, 32, 72}, {43, 83, 63, 3},
 6.                  {24, 94, 54, 84}, {15, 65, 85, 5},
 7.                  {36, 26, 66, 56}, {97, 17, 37, 7}};
 8.         Thread t1;
 9.         for(int i = 0; i < odds.length; i++){
10.             t1 = new SyncProblems(odds[i]);
11.             t1.start();
12.         }
13.     }
14.     int a [];
15.     public SyncProblems(int a []){
16.         this.a = a;
17.     }
18.     public void run(){
19.         PrintingUtils.print(a);
20.     }
21. }
```

- Although the original intent was to display each array sorted, the output of the above application is shown below.

```
1,2,3,4,223245267,,,,,,324354153617,,,,,,526384655637,
,,5,,,7283,946685

6

,7,8,9
97
```

- The next page introduces the `synchronized` keyword as a means of correcting the above problem.

Synchronization

- The problem in the previous code was that multiple threads where inside of the `PrintingUtils.print` method at the same time, each competing for access to the standard output `System.out` to print the array.

- To guard against this, every object in Java has a lock with which it is associated.
  - When an object is locked by one thread, and another thread tries to call a `synchronized` method or `synchronized` block on the same object, the second thread will block until the object is unlocked.
  - The portion of the code that is `synchronized` is often referred to as a **critical section**.
  - The example below is a revised version of the previous `PrintingUtils.java` where the `print` method has been `synchronized`.

PrintingUtils2.java

```
 1. package examples.threads;
 2. import java.util.*;
 3. public class PrintingUtils2 {
 4.     public static synchronized void print(int x[]){
 5.         Arrays.sort(x);
 6.         for(int i = 0; i < x.length; i++){
 7.             System.out.print(x[i]);
 8.             if(i < x.length - 1)
 9.                 System.out.print(",");
10.         }
11.         System.out.println();
12.     }
13. }
```

  - The application to test the new version of the `print` method is shown on the next page.

Synchronization

NoSyncProblems.java

```
 1. package examples.threads;
 2. public class NoSyncProblems extends Thread{
 3.     public static void main(String args[]){
 4.         int odds [][] = {{9, 8, 7, 6, 5, 4, 3, 2, 1},
 5.                 {52, 22, 32, 72}, {43, 83, 63, 3},
 6.                 {24, 94, 54, 84}, {15, 65, 85, 5},
 7.                 {36, 26, 66, 56}, {97, 17, 37, 7}};
 8.         Thread t1;
 9.         for(int i = 0; i < odds.length; i++){
10.             t1 = new NoSyncProblems(odds[i]);
11.             t1.start();
12.         }
13.     }
14.     int a [];
15.     public NoSyncProblems(int a []){
16.         this.a = a;
17.     }
18.     public void run(){
19.         PrintingUtils2.print(a);
20.     }
21. }
```

- The output generated by the above application is shown below.

```
1,2,3,4,5,6,7,8,9
22,32,52,72
3,43,63,83
24,54,84,94
5,15,65,85
26,36,56,66
7,17,37,97
```

- Although synchronizing the method resulted in the desired output (the sorting process inside of the method was never really a problem), only the actual printing of the results resulted in the inconsistent output.
  - For this reason, it would have been more efficient to make each thread block only for the printing process - not the sorting.

Synchronization

- A synchronized block of code can be used to surround a critical region of code, rather than the entire method.

- The example below shows a new version of the `PrintingUtils` that only synchronizes the printing, allowing each thread to at least be sorted prior to being blocked.

PrintingUtils3.java

```
 1. package examples.threads;
 2. import java.util.*;
 3. public class PrintingUtils3 {
 4.     static Object o = new Object();
 5.     public static void print(int x[]){
 6.         Arrays.sort(x);
 7.         synchronized (o){
 8.             for(int i = 0; i < x.length; i++){
 9.                 System.out.print(x[i]);
10.                 if(i < x.length - 1)
11.                     System.out.print(",");
12.             }
13.             System.out.println();
14.         }
15.     }
16. }
```

- The code above obtains the lock associated with the `Object o` to synchronize the critical region.
- Since the `print` method itself is no longer `synchronized`, each thread is able to enter the method and execute the `sort` method prior to being blocked by another thread that may have already entered the `synchronized` block of code.
- When a thread leaves the `synchronized` block of code, the lock it holds is relinquished, and another thread is able to enter the region and obtain the lock from `Object o`.

Performance Issues

- While threads generally improve program performance, you can unintentionally slow the program by using threads and the `synchronized` keyword incorrectly.

- Synchronizing is an expensive operation. It can considerably inhibit program performance. You should synchronize access to data that is shared between threads. If it is not shared, there is no reason to synchronize access to it.

- If only a small part of the method accesses the shared data, synchronize the block around an object instead of the entire method. The more asynchronous activity you can allow, the faster the program will run.

- Note also that shared data need not be synchronized unless you can write into the data.

- Thread starvation is caused when one or more threads do not have a chance to run because other threads monopolize the CPU. This can be prevented (for same-priority threads) by using `yield()` to give other threads a chance to run.

- Finally, do not forget to use `notify()` if you change a condition upon which other threads are waiting (using `wait()`).

Exercises

1. Write a multi-threaded version of the Echo service application.
   - The server should be able to handle multiple clients.
   - This server should sit in a loop and create a separate thread each time a client connects.

Chapter 13:
Remote Method Invocation (RMI)

1) Introduction 13-2

2) RMI Architecture 13-3

3) The Remote Interface 13-4

4) The Remote Object 13-5

5) Writing the Server 13-6

6) The RMI Compiler 13-8

7) Writing the Client 13-9

8) Remote Method Arguments and Return Values 13-10

9) Dynamic Loading of Stub Classes 13-11

10) Remote RMI Client Example 13-12

11) Running the Remote RMI Client Example 13-20

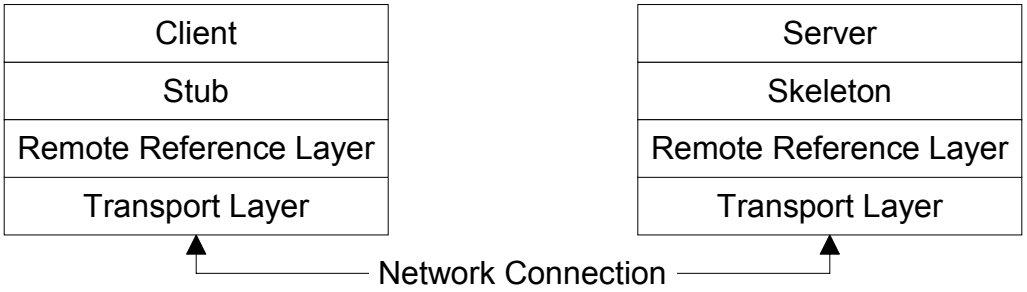
Introduction

- The **R**emote **M**ethod **I**nvocation (**RMI**) model represents a **distributed object application.**
  - RMI allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client.
    - Therefore, RMI implies a client and a server.
- The server application typically creates an object and makes it accessible remotely.
  - Therefore, the object is referred to as a remote object.
  - The server registers the object that is available to clients.
    - One of the ways this can be accomplished is through a naming facility provided as part of the JDK, which is called the `rmiregistry`.
    - The server uses the registry to bind an arbitrary name to a remote object.
- A client application receives a reference to the object on the server and then invokes methods on it.
  - The client looks up the name in the registry and obtains a reference to an object that is able to interface with the remote object.
    - The reference is referred to as a remote object reference.
  - Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

RMI Architecture

- The interface that the client and server objects use to interact with each other is provided through stubs/skeleton, remote reference, and transport layers.
  - Stubs and skeletons are Java objects that act as proxies to the client and server, respectively.
    - All the network-related code is placed in the stub and skeleton, so that the client and server will not have to deal with the network and sockets in their code.
  - The remote reference layer handles the creation of and management of remote objects.
  - The transport layer is the protocol that sends remote object requests across the network.

- A simple diagram showing the above relationships is shown below.



The Remote Interface

- The server's job is to accept requests from a client, perform some service, and then send the results back to the client.
  - The server must specify an interface that defines the methods available to clients as a service.
    - This **remote interface** defines the client view of the remote object.
- The remote interface is always written to extend the `java.rmi.Remote` interface.
  - `Remote` is a "marker" interface that identifies interfaces whose methods may be invoked from a non-local virtual machine.

CalendarTask.java

```
1.  package examples.rmi;
2.  import java.rmi.Remote;
3.  import java.rmi.RemoteException;
4.  import java.util.Calendar;
5.  public interface CalendarTask extends Remote {
6.      Calendar getDate() throws RemoteException;
7.  }
```

- In the example above, `getDate()` is a remote method of the remote interface `CalendarTask`.
  - All methods defined in the remote interface are required to state that they throw a `RemoteException`.
    - A `RemoteException` represents communication-related exceptions that may occur during the execution of a remote method call.

The Remote Object

- An implementation of the `CalendarTask` interface is shown below.
  - The implementation is referred to as the remote object.
  - The implementation class extends `UnicastRemoteObject` to link into the RMI system.
    - This is not a requirement. A class that does not extend `UnicastRemoteObject` may use its `exportObject()` method to be linked into RMI.
  - When a class extends `UnicastRemoteObject`, it must provide a constructor declaring that it may throw a `RemoteException` object.
    - When this constructor calls super(), it activates code in `UnicastRemoteObject`, which performs the RMI linking and remote object initialization.

CalendarImpl.java

```
 1. package examples.rmi;
 2. import java.rmi.RemoteException;
 3. import java.rmi.server.UnicastRemoteObject;
 4. import java.util.Calendar;
 5.
 6. public class CalendarImpl extends UnicastRemoteObject
 7.          implements CalendarTask {
 8.
 9.     private int counter = 1;
10.
11.     public CalendarImpl() throws RemoteException {}
12.
13.     public Calendar getDate() throws RemoteException{
14.         System.out.print("Method called on server:");
15.         System.out.println("counter = " + counter++);
16.         return Calendar.getInstance();
17.     }
18. }
```

Writing the Server

- The server creates the remote object, registers it under some arbitrary name, then waits for remote requests.
  - The `java.rmi.registry.LocateRegistry` class allows the RMI registry service (provided as part of the JVM) to be started within the code by calling its `createRegistry` method.
    - This could have also been achieved by typing the following at a command prompt: `rmiregistry`.
    - The default port for RMI is `1099`.
  - The `java.rmi.registry.Registry` class provides two methods for binding objects to the registry.
    - `Naming.bind("`*`ArbitraryName`*`", `*`remoteObj`*`);` throws an Exception if an object is already bound under the "*`ArbitrayName`*. "
    - `Naming.rebind ("`*`ArbitraryName`*`", `*`remoteObj`*`);` binds the object under the "*`ArbitraryName`*" if it does not exist or overwrites the object that is bound.
- The example on the following page acts as a server that creates a `CalendarImpl` object and makes it available to clients by binding it under a name of "`TheCalendar`. "

Writing the Server

CalendarServer.java

```
 1.  package examples.rmi;
 2.  import java.rmi.Naming;
 3.  import java.rmi.registry.LocateRegistry;
 4.
 5.  public class CalendarServer {
 6.
 7.      public static void main(String args[]) {
 8.          System.out.println("Starting server...");
 9.          // Start RMI registry service and bind
10.          // object to the registry
11.          try {
12.              LocateRegistry.createRegistry(1099);
13.              Naming.rebind("TheCalendar",
14.                      new CalendarImpl());
15.          } catch (Exception e) {
16.              e.printStackTrace();
17.              System.exit(1);
18.          }
19.          System.out.println("Server ready");
20.      }
21.  }
```

- If both the client and the server are running Java SE 5 or higher, no additional work is needed on the server side.
  - Simply compile the `CalendarTask`, `CalendarImpl`, and `CalendarServer`, and the server can then be started.
  - The reason for this is the introduction in Java SE 5 of dynamic generation of stub classes.
    - Java SE 5 adds support for the dynamic generation of stub classes at runtime, eliminating the need to use the RMI stub compiler, `rmic`, to pre-generate stub classes for remote objects.
    - Note that `rmic` must still be used to pre-generate stub classes for remote objects that need to support clients running on earlier versions.

The RMI Compiler

- If RMI is being used with a version of Java prior to Java SE 5, a stub must be generated on the server-side and made available to the client.
  - The RMI compiler (`rmic`) is a tool used to create any necessary stubs and/or skeletons to support the remote object.
    - Skelton(s) have been optional since Java SE 1.2.
  - The `rmic` compiler is passed to the compiled version of the remote object and generates a stub class from it as shown below.

```
rmic -keep -d %CLASSES% examples.rmi.CalendarImpl
```

    - The "`-keep`" is optional and is being used so that, in addition to the generated `.class` files, the `.java` files will be retained so that they can be viewed if desired.
    - The result of running the `rmic` command above is a file named `CalendarImpl_Stub.class`.
    - Since the "`-keep`" option was used, there is also a file named `CalendarImpl_Stub.java`.
  - The `CalendarImpl_Stub` class generated is a client-side component and as such, must exist on the client's classpath in order for client code to successfully communicate with the server.
    - If the stub class is not available locally to the client, it must be loaded dynamically over the network.
    - Keep in mind that this is only necessary if a version of Java prior to Java SE 5 is being used.

Writing the Client

- An RMI client is a program that accesses the services provided by a remote object.
  - The `java.rmi.registry.LocateRegistry` class allows the RMI registry service to be located by a client by its `getRegistry` method.
    - The `java.rmi.registry.Registry` class provides a `lookup` method that takes the "`ArbitraryName`" the remote object was bound to by the server.
- Once the client obtains a reference to a remote object, it invokes methods as if the object were local.

CalendarClient.java

```
 1.  package examples.rmi;
 2.
 3.  import java.rmi.registry.*;
 4.  import java.util.Calendar;
 5.
 6.  public class CalendarClient {
 7.
 8.      public static void main(String args[]) {
 9.          Calendar c = null;
10.          CalendarTask remoteObj;
11.          String host = "localhost";
12.          if(args.length == 1)
13.              host = args[0];
14.          try {
15.              Registry r =
16.                LocateRegistry.getRegistry(host, 1099);
17.              Object o = r.lookup("TheCalendar");
18.              remoteObj = (CalendarTask) o;
19.              c = remoteObj.getDate();
20.          } catch (Exception e) {
21.              e.printStackTrace();
22.          }
23.          System.out.printf("%tc", c);
24.      }
25.  }
```

Remote Method Arguments and Return Values

- The arguments to a remote method must be `Serializable`.
  - They must be primitive types or objects that implement the `Serializable` interface.
  - The same restriction applies to return values.
- The RMI stub/skeleton layer decides how to send arguments and return values over the network.
  - If the object is `Serializable` but not `Remote`:
    - the object is serialized and streamed in byte format; and
    - the receiver de-serializes the bytes into a copy of the original object.
  - If the object is a `Remote` object:
    - a remote reference for the object is marshaled and sent to the remote process; and
    - this reference is received and converted into a stub for the original object.
  - If the argument or return value is not serializable, a `java.rmi.MarshalException` is thrown.
- The key difference between remote and non-remote objects is that `Remote` objects are sent by reference, while non-remote objects (and primitive types) are sent by copy.

Dynamic Loading of Stub Classes

- If the client stub class is not available in the local `CLASSPATH`, it must be loaded dynamically over the network.
  - This is a typical scenario when the client and server are not running on the same machine.
  - We will illustrate downloading of stub classes via a web server.
- When the RMI run-time system marshals a remote object stub, it encodes a URL in the byte stream to tell the process on the other end of the stream where to look for the class file for the marshaled object.
  - This URL is obtained from a system property called `java.rmi.server.codebase`.
    - We will set this property on the command line to point to a directory within the web server's document base.
  - Note that in order for a Java runtime system to be able to load classes remotely, it has to have a security manager installed that will allow the remote load.
    - There is one provided by the `java.rmi.RMISecurityManager` class.
  - The final issue is that the default Java security policy does not allow all the networking operations required to load a class from a remote host.
    - An RMI client that needs to load classes remotely must have a policy file granting the necessary permissions.
    - The name of the policy file can be specified on the command line by setting the `java.security.policy` property.

Remote RMI Client Example

- The RMI application shown below illustrates dynamic loading of stub classes.
  - It also shows an example of a `Remote` object used as a method argument.
  - Following the source code are detailed instructions on how to run the Client and Server.

- We begin with the remote interface.

Account.java

```
 1.  package examples.rmi;
 2.
 3.  import java.rmi.*;
 4.
 5.  public interface Account extends Remote {
 6.      public String getName() throws RemoteException;
 7.
 8.      public double getBalance()
 9.        throws RemoteException;
10.
11.      public void withdraw(double amt)
12.        throws RemoteException;
13.
14.      public void deposit(double amt)
15.        throws RemoteException;
16.
17.      public void transfer(double amt, Account src)
18.        throws RemoteException;
19.  }
```

- The implementation of the remote interface is shown on the next page.

Remote RMI Client Example

AccountImpl.java

```
 1. package examples.rmi;
 2.
 3. import java.rmi.server.*;
 4. import java.rmi.*;
 5.
 6. public class AccountImpl extends UnicastRemoteObject
 7.   implements Account {
 8.     private double balance = 0.0;
 9.     private String name = "";
10.
11.     public AccountImpl(String aName)
12.       throws RemoteException {
13.         name = aName;
14.     }
15.
16.     public String getName() throws RemoteException {
17.         return name;
18.     }
19.
20.     public double getBalance()
21.       throws RemoteException {
22.         return balance;
23.     }
24.
25.     public void withdraw(double amt)
26.       throws RemoteException {
27.         if (amt > balance)
28.             throw new RemoteException();
29.         balance -= amt;
30.     }
31.
32.     public void deposit(double amt)
33.       throws RemoteException {
34.         balance += amt;
35.     }
36.
37.     public void transfer(double amt, Account src)
38.       throws RemoteException {
39.         src.withdraw(amt);
40.         this.deposit(amt);
41.     }
42. }
```

Remote RMI Client Example

- The Server is shown below with the following features.
  - The compiling of the stub class for the client is done using `Runtime.exec()`.
    - The `exec` method allows the JVM to run an external process (in this case the rmi compiler - `rmic`).
  - The server creates and starts the registry service.

AccountServer.java

```
 1. package examples.rmi;
 2.
 3. import java.io.IOException;
 4. import java.net.InetAddress;
 5. import java.rmi.registry.*;
 6.
 7. public class AccountServer {
 8.     private static String buildCommandLine(){
 9.         String jcp = "java.class.path";
10.         StringBuffer sb = new StringBuffer();
11.         sb.append('"');
12.         sb.append(System.getProperty(jcp));
13.         sb.append('"');
14.         String classpath = sb.toString();
15.         sb.setLength(0);
16.         sb.append("rmic -d ").append(classpath);
17.         sb.append(" -classpath ").append(classpath);
18.         sb.append(" examples.rmi.AccountImpl");
19.         System.out.println(sb.toString());
20.         return sb.toString();
21.     }
22.     public static void main(String args[]) {
23.         // execute rmic as an external process
24.         Process p = null;
25.         try {
26.             String command = buildCommandLine();
27.             p = Runtime.getRuntime().exec(command);
28.             p.waitFor(); // wait for completion
29.         } catch (Exception e1) {
30.             e1.printStackTrace();
31.         }
```

Remote RMI Client Example

AccountServer.java - continued

```
32.
33.          try {
34.              String key = "java.rmi.server.codebase";
35.              InetAddress server =
36.                  InetAddress.getLocalHost();
37.              String address = server.getHostAddress();
38.              String value =
39.                  "http://" + address + ":8080/";
40.              System.setProperty(key, value);
41.              //Start the StubServer
42.              Thread t = new StubServer();
43.              t.start();
44.              // Create registry service
45.              Registry reg =
46.                  LocateRegistry.createRegistry(1099);
47.              // Create some Accounts
48.              AccountImpl acct1 =
49.                new AccountImpl("Alan");
50.              AccountImpl acct2 =
51.                new AccountImpl("Dave");
52.
53.              // Register with the naming registry.
54.              reg.rebind("Alan", acct1);
55.              reg.rebind("Dave", acct2);
56.
57.              System.out.println("Accts registered");
58.          } catch (Exception e) {
59.              e.printStackTrace();
60.          }
61.      }
62.  }
```

Remote RMI Client Example

- Although a web server such as Tomcat, WebLogic, or WebSphere could be used to host the stub class necessary for dynamic loading of the stub class, the file below is a simple web server based on the code from the Networking chapter of this course.

StubServer.java

```
 1. package examples.rmi;
 2.
 3. import java.io.*;
 4. import java.net.*;
 5.
 6. public class StubServer extends Thread {
 7.
 8.     static byte[] hdrNotFound =
 9.         "HTTP/1.0 404 Not Found\n\n".getBytes();
10.     static byte[] notFound =
11.         "<html>Resource Not Found</html>".getBytes();
12.     static byte[] hdrOK =
13.         "HTTP/1.0 200 OK\n\n".getBytes();
14.     static byte[] testResponse =
15.         "<html>Server operational</html>".getBytes();
16.
17.     public void run() {
18.         ServerSocket theServer = null;
19.         Socket clientSocket;
20.         // Attempt to start the server
21.         try {
22.             theServer = new ServerSocket(8080);
23.             while (true) {
24.                 clientSocket = theServer.accept();
25.                 handleClient(clientSocket);
26.             }
27.         } catch (IOException ioe) {
28.             ioe.printStackTrace();
29.             System.exit(1);
30.         }
31.     }
32.
```

▸ *Continued on following page*

Remote RMI Client Example

StubServer.java - *continued*

```
33.     private void handleClient(Socket cSocket) {
34.         OutputStream toClient = null;
35.         BufferedReader fromClient = null;
36.         try {
37.             // Get Input and Output
38.             fromClient = new BufferedReader(
39.                    new InputStreamReader(cSocket
40.                     .getInputStream()));
41.             toClient = cSocket.getOutputStream();
42.             // read from Client
43.             String theLine = fromClient.readLine();
44.             String request = theLine.split(" ")[1];
45.             System.out.println("StubServer Request:"
46.                                + theLine);
47.             if (request.equals("/")) {
48.                 toClient.write(hdrOK);
49.                 toClient.write(testResponse);
50.             } else {
51.                 processStub(request, toClient);
52.             }
53.
54.             fromClient.close();
55.             toClient.close();
56.             cSocket.close();
57.         } catch (IOException ioe) {
58.             String msg = "Connection lost";
59.             System.out.println(msg);
60.         }
61.     }
```

▸ *Continued on following page*

Remote RMI Client Example

StubServer.java - *continued*

```
62.     private void processStub(String req,
63.             OutputStream toClient){
64.         InputStream is =
65.             this.getClass().getResourceAsStream(req);
66.
67.         byte[] bufferedStub = null;
68.         try {
69.             if (is != null) {
70.                 int size = is.available();
71.                 bufferedStub = new byte[size];
72.                 is.read(bufferedStub);
73.                 is.close();
74.                 toClient.write(hdrOK);
75.                 toClient.write(bufferedStub);
76.             } else {
77.                 toClient.write(hdrNotFound);
78.                 toClient.write(notFound);
79.             }
80.         } catch (IOException e) {
81.             e.printStackTrace();
82.         }
83.     }
84. }
```

- Finally, the client code is shown below.

AccountClient.java

```
 1. package examples.rmi;
 2.
 3. import java.net.URL;
 4. import java.rmi.*;
 5. import java.rmi.registry.*;
 6.
 7. public class AccountClient {
 8.     public static void main(String args[]) {
 9.         String host = "localhost";
10.         if (args.length > 0) { host = args[0]; }
11.         try {
12.             String key = "java.security.policy";
13.             Class c = AccountClient.class;
14.             URL u =c.getResource("policy.client");
```

Remote RMI Client Example

AccountClient.java - *continued*

```
15.             String value = u.getPath();
16.             System.setProperty(key, value);
17.             System.setSecurityManager(
18.                     new RMISecurityManager());
19.             Registry r =
20.               LocateRegistry.getRegistry(host, 1099);
21.             // Lookup Account objects
22.             Account act1 =
23.                 (Account) r.lookup("Alan");
24.             Account act2 =
25.                 (Account) r.lookup("Dave");
26.
27.             showBalance(act1);
28.             showBalance(act2);
29.
30.             // Make some deposits
31.             act1.deposit(200);
32.             act2.deposit(100);
33.
34.             // Show results
35.             System.out.println("Deposit 200 & 100");
36.             showBalance(act1);
37.             showBalance(act2);
38.
39.             // Do a transfer
40.             act2.transfer(10, act1);
41.
42.             // Show results
43.             System.out.println("Transfer 10");
44.             showBalance(act1);
45.             showBalance(act2);
46.         } catch (Exception e){ e.printStackTrace(); }
47.     }
48.
49.     public static void showBalance(Account acct)
50.       throws RemoteException {
51.         System.out.println("Balance for " +
52.                 acct.getName() + " is " +
53.                 acct.getBalance());
54.     }
55. }
```

Running the Remote RMI Client Example

- Running the `AccountServer` will accomplish the following.
  - It generates the `AccountImpl_Stub` needed by the client so that it can be dynamically loaded by the client.
  - It starts the `StubServer` so that the `AccountImpl_Stub` is available via the `java.rmi.server.codebase` property.
  - It starts the rmi registry to bind the remote `Account` objects.
- Running the `AccountClient` will accomplish the following.
  - It installs the `RMISecurityManager`.
  - It utilizes the policy file named `policy.client` to allow the JVM to load a class from a remote URL.
  - If the Stub class is not available on the client's classpath when the lookup is performed, the client will automatically retrieve the necessary stub from the servers codebase.

Exercises

1. Build and test an implementation for a `MathServices` interface with remote methods as shown below.

```
public double sqroot (double value);
public double square(double value);
```

   - Begin with `MathServices.java` in the `starters` directory.

This Page Intentionally Left Blank

Chapter 14:
Java Database Connectivity (JDBC)

1) Introduction..... 14-2

2) Relational Databases..... 14-4

3) Structured Query Language..... 14-5

4) A Sample Program..... 14-6

5) Transactions..... 14-9

6) Meta Data..... 14-15

Introduction

- The **J**ava **D**ata**B**ase **C**onnectivity (JDBC) standard allows Java applets and applications to access data from all the major database management systems with a single API. JDBC is modeled after Microsoft's **O**pen **D**ata**B**ase **C**onnectivity (ODBC) standard. Developers familiar with ODBC will be immediately proficient with the JDBC.

- Before we look at some examples, we will briefly review the concept of tables, relational databases, and the Structured Query Language (SQL).

- A table consists of two parts – its `schema` and the `data` it contains. The schema of a table describes the name and type of data fields it can store. For example, a table named `Books` may have the following schema.

`Books`

Field Name	Data Type
BookId	`Integer`
Title	`Character (20)`
Author	`Character (30)`
NumPages	`Integer`

  - Although not shown above, the schema of a table also contains information on indices, which may be used to:
    - link tables together;
    - enforce constraints on a data field or set of fields (e.g. keep them unique); and
    - speed data access.

Introduction

- Data contained by a table is stored in units called records. Each record contains data for each field defined in the table's schema. The table below shows the data stored in the `Books` table. The table contains four records. Each record stores data as defined by the table's schema.

`Books`

BookId	Title	Author	NumPages
1	AirFrame	Michael Crichton	347
2	Green Eggs and Ham	Dr. Seuss	62
3	The Vampire Lestat	Anne Rice	550
4	Chaos: Making a New Science	James Gleick	317

Relational Databases

- Database theory has undergone several milestones and models. Most modern databases are relational as opposed to hierarchical or networked (having nothing to do with computer networks).

- Relational databases allow the user to specify relationships between one or more tables in the database.

- Using an SQL statement, you can recall data fields from more than one table using a relationship between the tables.

- Designing database tables and the relationships between them is a difficult task.

- Imagine that we have a table called `Libraries` that stores the library name, a `bookId`, and the number of copies for a particular book. The `bookId` in this table can relate to the `Books` table defined previously.

- In order to find the names and number of copies for each library book, we must link the tables. Since the `bookId` field is the same in each table, it will be used as the linking criteria. Below is an SQL statement that will recall data from both the `Libraries` table and the `Books` table.

  SELECT Libraries.LibraryName, Books.Title, Libraries.NumBooks
  FROM Libraries, Books WHERE Libraries.BookId = Books.BookId

Structured Query Language

- The **S**tructured **Q**uery **L**anguage (SQL) is a standard for accessing and updating relational databases. To use SQL in Java, one must create an SQL statement as a `String` and then feed the `String` to the database engine.

- JDBC requires drivers to support the ANSI SQL-92 standard to be "JDBC compliant."
  - The basic units of SQL are tables, columns, and rows.
  - The "relation" is mapped in the table via a correspondence between a row (a record) and a column (a piece of data with a name).
  - SQL statements are normally used to recall and update data in a database, but they can also be used to create and delete tables, create and delete indices, and perform other system database functions.

- Below are examples of SQL statements.

```
SELECT BookId, Title FROM Books
DELETE FROM Books WHERE BookId = 1
```

  - The first statement is a `SELECT` statement and is used to get data from the database. The SQL keywords are in capital letters.
  - The second statement will delete the records in table `Books`, where `BookId` equals `one`.

A Sample Program

SimpleQuery.java

```
 1. package examples.jdbc;
 2. import java.sql.*;
 3. public class SimpleQuery {
 4.     public static void main(String args[]) {
 5.         try {
 6.          //Load the database driver
 7.          String s =
 8.               "sun.jdbc.odbc.JdbcOdbcDriver";
 9.             Class.forName(s);
10.             // Establish connection
11.             String url = "jdbc:odbc:ExampleDb";
12.             Connection con =
13.               DriverManager.getConnection(url);
14.             // Create and send statement
15.             String query = "SELECT Author,Title, " +
16.                     "NumPages FROM BOOKS";
17.             Statement stmt = con.createStatement();
18.             ResultSet rslt =
19.               stmt.executeQuery(query);
20.             // Process data
21.             String author, title;
22.             int pages;
23.             while (rslt.next()) {
24.                 author = rslt.getString("Author");
25.                 title = rslt.getString("Title");
26.                 pages = rslt.getInt("NumPages");
27.                 System.out.println(title + " by " +
28.                     author + " has " + pages +
29.                     " pages.");
30.             }
31.             // Cleanup
32.             rslt.close();
33.             stmt.close();
34.             con.close();
35.         } catch (SQLException sqle){
36.             System.out.println(sqle.getMessage());
37.         } catch (ClassNotFoundException e) {
38.          e.printStackTrace();
39.     }
40.     }
41. }
```

A Sample Program

- The general idea behind all JDBC applications may be summarized as sending SQL statements to the database driver and receiving back either a status (for updates, inserts or deletes) or a result set.

- Load the JDBC driver.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

  - This expression will load the JDBC-ODBC bridge class, which ships as part of the JDK.

- Create a connection to the database driver. The `Connection` interface specifies the interface to the JDBC driver.

```
String url = "jdbc:odbc:ExampleDb";
Connection con =
    DriverManager.getConnection(url);
```

- Create a Statement object.

```
Statement stmt = con.createStatement();
```

- Receive a `ResultSet` object from the driver. When the statement is executed, a `ResultSet` object is returned (if the statement is a Query). It consists of a number of rows, each row having some number of columns.

```
ResultSet rslt;
    rslt=stmt.executeQuery(sQuery);
```

A Sample Program

- Access the rows and individual values of each row (columns).
  - The `ResultSet` object has numerous `getXXX()` methods, where `XXX` is a data type.
  - There are two types of the `getXXX()` methods, one that takes a column name and another that takes a column number (with column numbers beginning with 1).

```
while (rslt.next()) {
    author = rslt.getString("Author");
    title = rslt.getString("Title");
    pages = rslt.getInt("NumPages");
    System.out.println(title + " by " +
        author + " has " + pages +
        " pages.");
}
```

- Close the result set, statement, and connection. This allows resources to be freed in both the JVM and the database.

```
rslt.close();
stmt.close();
con.close();
```

- Catch exceptions that are thrown by the driver. Note that SQL exceptions may be chained together to send multiple error messages back to the user.
  - The `getNextException()` method is used to get the next exception in the chain.
  - The `getSQLState()`, `getMessage()`, and `getErrorCode()` methods all provide more detailed information about the error.

Transactions

- The previous example demonstrated how to make a simple query. In this section, we concentrate on how to make a transaction.

- A transaction allows multiple SQL statements to be grouped together in a manner such that, if all the statements complete successfully, any changes made to the database will be made permanently. If any of the statements in the transaction fail, all changes made to the database during the current transaction will be undone. The act of "undoing" parts of a transaction is called "**rollback**."

- In the following example, `AddBook.java`, information is added to two different tables. If the set of actions is invalid (perhaps because of bad data), the statement would fail and trigger a rollback on the database. In other words, if both inserts do not succeed, we want none of them to succeed.

Transactions

AddBook.java

```
 1. package examples.jdbc;
 2.
 3. import java.sql.*;
 4.
 5. public class AddBook {
 6.     static Connection conn;
 7.     static Statement stmt;
 8.     public static void main(String args[]) {
 9.         try {
10.             checkArguments(args);
11.             configureConnection();
12.             configureTransactions();
13.             stmt = conn.createStatement();
14.             // begin transaction
15.             conn.setAutoCommit(false);
16.             manipulateData(args);
17.             conn.commit(); // commit transaction
18.             // Cleanup
19.             stmt.close();
20.             conn.close();
21.         } catch (Exception e) {
22.             e.printStackTrace();
23.         }
24.     }
25.
26.     public static void checkArguments(String args[]){
27.         if (args.length != 3) {
28.             String msg = "USAGE: java AddBook "
29.                 + "<author> <title> <pages>";
30.             throw new IllegalArgumentException(msg);
31.         }
32.     }
33.
34.     public static void configureConnection() throws
35.             ClassNotFoundException, SQLException{
36.         String s ="sun.jdbc.odbc.JdbcOdbcDriver";
37.         Class.forName(s);
38.         String url = "jdbc:odbc:ExampleDb";
39.         conn = DriverManager.getConnection(url);
40.     }
41.
```

Transactions

AddBook.java - *continued*

```
42.     public static void configureTransactions()
43.             throws SQLException{
44.         DatabaseMetaData dbmd = conn.getMetaData();
45.         if (!dbmd.supportsTransactions()) {
46.             String msg = "No Transaction Support";
47.             throw new RuntimeException(msg);
48.         }
49.         int txLevel =
50.             conn.getTransactionIsolation();
51.         // try to set a higher TRANSACTION LEVEL
52.         if (txLevel == Connection.TRANSACTION_NONE) {
53.             int txRC =
54.                Connection.TRANSACTION_READ_COMMITTED;
55.             boolean readCommitted = dbmd.
56.              supportsTransactionIsolationLevel(txRC);
57.             if (readCommitted)
58.                 conn.setTransactionIsolation(txRC);
59.         }
60.     }
61.     public static void manipulateData(String args[])
62.             throws SQLException{
63.         String sql;
64.         ResultSet rslt;
65.         try {
66.             // Get new AuthorId
67.             sql= "SELECT MAX(AuthorId) AS " +
68.                  "Max_AuthorId FROM AUTHORS";
69.             rslt = stmt.executeQuery(sql);
70.             int authID = 0;
71.             while (rslt.next())
72.                 authID =
73.                     rslt.getInt("Max_AuthorId") + 1;
74.             // Add to Authors
75.             sql = "INSERT INTO AUTHORS " +
76.                   "(AuthorId, Author) VALUES (" +
77.                   authID + ", '" + args[0] + "')";
78.             stmt.executeUpdate(sql);
79.             // Get new BookId
80.             sql = "SELECT MAX(BookId) AS " +
81.                   "Max_BookId FROM Books_II";
82.             rslt = stmt.executeQuery(sql);
```

Transactions

AddBook.java - *continued*

```
 83.              int bookID = 0;
 84.
 85.              while (rslt.next())
 86.                  bookID =
 87.                      rslt.getInt("Max_BookId") + 1;
 88.              // Add to Books_II
 89.              sql = "INSERT INTO BOOKS_II " +
 90.                  "(BookId, AuthorId, Title, NumPages)"
 91.                  + " VALUES (" + bookID + ", "
 92.                   + authID + ", '" + args[1] + "', "
 93.                  + args[2] + ")";
 94.              stmt.executeUpdate(sql);
 95.
 96.          } catch (SQLException e) {
 97.              // rollback transaction
 98.              conn.rollback();
 99.              throw new SQLException(e.getMessage());
100.          }
101.      }
102.  }
```

- In real life, the interface to a database operation would probably be through a GUI, but to make things simple, we execute the following.

```
java examples.chapter8.AddBook Heinlein "To Sail Beyond
The Sunset" 425
```

Transactions

- Since no validation is performed on the data, an improper value can cause one of the insert statements to fail.
  - The first item to note is a reference to a new interface, `DatabaseMetaData`, which provides information about the database itself, such as support for transactions, information on catalogs, primary keys, etc.
  - Here we use `DatabaseMetaData` to see if the database to which we are connected supports transactions.

```
DatabaseMetaData dbmd = conn.getMetaData();
if (!dbmd.supportsTransactions()) {
    String msg = "No Transaction Support";
    throw new RuntimeException(msg);
}
```

- If the database does support transactions, we want to ensure that the current connection has an appropriate Transaction Isolation Level. First we ask the database whether it will support a particular isolation level, then we set the current connection to that level.

```
int txLevel =
    conn.getTransactionIsolation();
// try to set a higher TRANSACTION LEVEL
if (txLevel == Connection.TRANSACTION_NONE) {
    int txRC =
        Connection.TRANSACTION_READ_COMMITTED;
    boolean readCommitted = dbmd.
    supportsTransactionIsolationLevel(txRC);
    if (readCommitted)
        conn.setTransactionIsolation(txRC);
}
```

- Now we begin to perform transactions.

Transactions

- We must turn the `AutoCommit` feature off or every statement will be committed to the database immediately after it executes.
  - Using the feature, any statements executed will act as a transaction until either `commit()` or `rollback()` is called.
  - Once either of these methods has been called, a new transaction will begin for subsequent database updates.

```
conn.setAutoCommit(false);
```

- The next statement demonstrates a database update.
  - Rather than using the `executeQuery()` method, we use the `executeUpdate()` method, which returns the number of rows affected by the statement.

```
sql = "INSERT INTO AUTHORS " +
    "(AuthorId, Author) VALUES (" +
    authID + ", '" + args[0] + "')";
stmt.executeUpdate(sql);
```

- The `commit()` statement will end the transaction and make permanent changes to the database.

```
conn.commit();
```

- If an error occurs, an exception is thrown so the `rollback()` method may be called to restore the database to its original state.

```
conn.rollback();
```

Meta Data

- Meta data is data about data. In addition to allowing the user to access and update data in the database, JDBC also provides a way to analyze the schema of a table and the schema of a database.

- Two interfaces are provided to give you access to the schemas.

  - `ResultSetMetaData` allows the user to examine the properties and columns for a `ResultSet`.

    - A `ResultSetMetaData` object may be returned with a call to `getMetaData()` in the `ResultSet` interface.
    - Methods are available for determining the properties of the result set as well as examining the columns in the result set.

  - `DatabaseMetaData` allows the user to examine the properties and tables of the database.

    - A `DatabaseMetaData` object may be returned with a call to `getMetaData()` in the `Connection` interface.
    - The `getTables()` method will return a `ResultSet` with a description of all the tables in the database.
    - There are many other methods available to access specific database properties.

Meta Data

SimpleMetaData.java

```
 1. package examples.jdbc;
 2.
 3. import java.sql.*;
 4.
 5. public class SimpleMetaData {
 6.     static {
 7.         try {
 8.             // Load database driver
 9.             String s ="sun.jdbc.odbc.JdbcOdbcDriver";
10.             Class.forName(s);
11.         } catch (ClassNotFoundException cnfe) {
12.             cnfe.printStackTrace();
13.             System.exit(0);
14.         }
15.     }
16.
17.     public static void main(String args[]) {
18.         if (args.length < 1) {
19.             String usage = "USAGE: java " +
20.                "examples.jdbc.SimpleMetaData "+
21.                "<tablename>";
22.             System.out.println(usage);
23.             System.exit(1);
24.         }
25.         try{
26.             displayInfo(args[0]);
27.         }catch (SQLException e) {
28.             StringBuffer sb = new StringBuffer();
29.             while (e != null) {
30.                 sb.append("SQLState: ");
31.                 sb.append(e.getSQLState());
32.                 sb.append("Message:  ");
33.                 sb.append(e.getMessage());
34.                 sb.append("ErrorCode:   ");
35.                 sb.append(e.getErrorCode());
36.                 sb.append("\n");
37.                 e = e.getNextException();
38.             }
39.             System.out.println(sb);
40.         }
41.     }
42.
```

Meta Data

SimpleMetaData.java - *continued*

```
43.     public static void displayInfo(String tName)
44.             throws SQLException{
45.         String tableName = tName;
46.
47.         // Establish connection
48.         String url = "jdbc:odbc:ExampleDb";
49.         Connection con =
50.             DriverManager.getConnection(url);
51.
52.         // Create and send statement
53.         String qry =
54.             "SELECT * FROM " + tableName;
55.
56.         Statement stmt = con.createStatement();
57.         ResultSet rslt = stmt.executeQuery(qry);
58.
59.         ResultSetMetaData rsmd = rslt.getMetaData();
60.         int colCount = rsmd.getColumnCount();
61.
62.         System.out.println("Field\tType");
63.         System.out.println("-----\t----");
64.         int width, type;
65.         for (int i = 1; i <= colCount; i++) {
66.             width = rsmd.getColumnDisplaySize(i);
67.             String colName = rsmd.getColumnName(i);
68.             type = rsmd.getColumnType(i);
69.
70.             String category = null;
71.             switch (type) {
72.             case Types.TINYINT:
73.             case Types.SMALLINT:
74.             case Types.INTEGER:
75.             case Types.BIGINT:
76.                 category = "Integer";
77.                 break;
78.             case Types.CHAR:
79.             case Types.VARCHAR:
80.                 category = "Char";
81.                 break;
82.             }
83.             System.out.print(colName + "\t" +
84.                              category);
```

Meta Data

SimpleMetaData.java - *continued*

```
85.                 if (category.equals("Char"))
86.                     System.out.print("(" + width + ")");
87.                 System.out.println();
88.             }
89.             // Cleanup
90.             rslt.close();
91.             stmt.close();
92.             con.close();
93.         }
94.     }
```

- When you run the program above, you will see the following.

```
java examples.jdbc.SimpleMetaData Books_II

Field       Type
-----       ----
BookId      Integer
AuthorId    Integer
Title       Char(30)
NumPages    Integer
```

Meta Data

- The code above gets the `ResultSetMetaData` object from the `ResultSet` object.

```
ResultSetMetaData rsmd = rslt.getMetaData();
```

  - Column information can then be obtained.

```
width = rsmd.getColumnDisplaySize(i);
String colName = rsmd.getColumnName(i);
type = rsmd.getColumnType(i);
```

Exercises

1. Create a command line Java program to query a database.
   - The database to query and the SQL statement are to be passed in from the command line.
   - The results should be printed on the standard output file (`System.out`).