Annotated Solution Guide for

Thinking in Java

Third Edition

Revision 1.0
April, 2004

Bruce Eckel
President, MindView, Inc.

About this document

This is the annotated solution guide for *Thinking in Java, 3rd edition* by Bruce Eckel. *Thinking in Java 3rd edition* is available in print form from Prentice Hall and freely downloadable from http://www.BruceEckel.com. This solution guide is only available electronically, as a PDF document (If it *were* in book form using the same format as *Thinking in Java*, this guide would be over 500 pages).

This guide also has some additional (supplemental) exercises which are not in *Thinking in Java* and for which solutions are not provided, as additional challenges and to provide exercises that may be used in teaching situations.

Chapter 1 has no exercises. The solutions for Chapter 2 and Chapter 3 are freely available (as a sample) as a PDF document here:

http://www.mindview.net/Books/TIJ/Solutions/

The remainder of the solutions are available, delivered electronically, for a fee of US 20$, only via credit card and only via the Internet at http://www.mindview.net/Books/TIJ/Solutions/orderForm.

Only a few exercises have been left to the reader (generally because the solution involves you performing some configuration on your own machine), including:

Chapter 4: Exercise 21

Chapter 5: Exercises 2, 3, 4, 7 & 9

Chapter 11: Exercises 40, 48 & 49

Chapter 14: Exercises 8, 9 & 24

Unpacking the distribution

This guide is distributed as a zipped distribution file containing this book in Adobe Acrobat PDF format, and a source-code tree in the **code.zip** file contained within the zipped distribution file. Please make sure your machine is equipped to unzip files before purchasing the guide (the zip format is virtually universal so you can almost always find a free utility for your platform if it isn't pre-installed; see the next paragraph). You must also install the Adobe Acrobat Reader, freely available for most platforms at http://www.adobe.com.

Linux/Unix Users: you must unzip the file using Info-Zip, which can be found for various platforms at http://www.info-zip.org/ or http://www.ibiblio.org/pub/packages/TeX/tools/info-zip/. This tool often comes pre-installed on Linux distributions. When you unzip the package for Linux/Unix, you must use the **–a** option to correct for the difference between DOS and Unix newlines, like this:

```
unzip –a TIJ3-solutions.zip
```

In the zip file that you receive upon purchasing this package, you will find a source-code directory tree that includes all the code listings shown in this guide, along with Ant build files to build all the listings using either the **javac** compiler from the Sun JDK (available at http://java.sun.com/j2se/) or the **jikes** compiler from IBM (available at http://oss.software.ibm.com/developerworks/opensource/jikes/) (note: **jikes** must be installed on top of the Sun JDK in order to work).

The build system used in this book is *Ant*, and you will see the Ant **build.xml** files in each of the subdirectories of the code tree. Ant is an open-source tool that migrated to the Apache project (available at http://ant.apache.org/. See that site for installation and configuration instructions). The Ant site contains the full download including the Ant executable and documentation. Ant has grown and improved until it is now generally accepted as the defacto standard build tool for Java projects. Refer to Chapter 15, *Discovering Problems* in *Thinking in Java 3rd* Edition for more details about Ant.

After you install and configure Ant on your computer, you can just type **ant** at the command prompt of the book's source-code root directory to build and test all the code.

Copyright

Please note that the entire *Thinking in Java Annotated Solution Guide* document is not freeware and so cannot be posted, resold or given away. It is copyrighted and sold only from the site www.BruceEckel.com.

Contents

Chapter 2

Exercise 1

```
//: c02:E01_HelloWorld.java
/****************** Exercise 1 ******************
 * Following the HelloDate.java example in this
 * chapter, create a "hello, world" program that
 * simply prints out that statement. You need
 * only a single method in your class (the "main"
 * one that gets executed when the program
 * starts). Remember to make it static and to
 * include the argument list, even though you
 * don't use the argument list. Compile the
 * program with javac and run it using java. If
 * you are using a different development
 * environment than the JDK, learn how to compile
 * and run programs in that environment.
 ***********************************************/
public class E01_HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello, world!");
  }
} ///:~
```

Exercise 2

```
//: c02:E02_ATypeName.java
/****************** Exercise 2 ******************
 * Find the code fragments involving ATypeName
 * and turn them into a program that compiles and
 * runs.
 ***********************************************/
public class E02_ATypeName {
  public static void main(String args[]) {
    E02_ATypeName a = new E02_ATypeName();
  }
} ///:~
```

Exercise 3

```
//: c02:E03_DataOnly.java
/****************** Exercise 3 ******************
 * Turn the DataOnly code fragments into a
 * program that compiles and runs.
 ***********************************************/
public class E03_DataOnly {
  int i;
  float f;
  boolean b;
  public static void main(String[] args) {
    E03_DataOnly d = new E03_DataOnly();
    d.i = 47;
    d.f = 1.1f;
    d.b = false;
  }
} ///:~
```

Exercise 4

```
//: c02:E04_DataOnly2.java
/****************** Exercise 4 ******************
 * Modify Exercise 3 so that the values of the
 * data in DataOnly are assigned to and printed
 * in main().
 ***********************************************/
public class E04_DataOnly2 {
  int i;
  float f;
  boolean b;
  public static void main(String[] args) {
    E04_DataOnly2 d = new E04_DataOnly2();
    d.i = 47;
    System.out.println("d.i = " + d.i);
    d.f = 1.1f;
    System.out.println("d.f = " + d.f);
    d.b = false;
    System.out.println("d.b = " + d.b);
  }
} ///:~
```

Exercise 5

```
//: c02:E05_Storage.java
/****************** Exercise 5 *****************
 * Write a program that includes and calls the
 * storage() method defined as a code fragment in
 * this chapter.
 ***********************************************/
public class E05_Storage {
  String s = "Hello, World!";
  int storage(String s) {
    return s.length() * 2;
  }
  void print() {
    System.out.println("storage(s) = " + storage(s));
  }
  public static void main(String[] args) {
    E05_Storage st = new E05_Storage();
    st.print();
  }
} ///:~
```

Exercise 6

```
//: c02:E06_StaticFun.java
/****************** Exercise 6 *****************
 * Turn the StaticFun code fragments into a
 * working program.
 ***********************************************/
class StaticTest {
  static int i = 47;
}
```

```
public class E06_StaticFun {
  static void incr() { StaticTest.i++; }
  public static void main(String[] args) {
    E06_StaticFun sf = new E06_StaticFun();
    sf.incr();
    E06_StaticFun.incr();
    incr();
  }
} ///:~
```

Notice that you can also just call **incr( )** by itself. This is because a **static** method (**main( )**, in this case) can call another **static** method without qualification.

Exercise 7

```
//: c02:E07_ShowArgs.java
// {Args: A B C}
/****************** Exercise 7 ******************
 * Write a program that prints three arguments
 * taken from the command line. To do this,
 * you'll need to index into the command-line
 * array of Strings.
 ***********************************************/
public class E07_ShowArgs {
  public static void main(String[] args) {
    System.out.println(args[0]);
    System.out.println(args[1]);
    System.out.println(args[2]);
  }
} ///:~
```

Note that you'll get a runtime exception if you run the program without enough arguments. You should actually test for the length of the array first, like this:

```
//: c02:E07_ShowArgs2.java
// {Args: A B C}
// Exercise 7B: Testing for the length of the array first.

public class E07_ShowArgs2 {
  public static void main(String[] args) {
    if(args.length < 3) {
      System.out.println("Need 3 arguments");
      System.exit(1);
    }
    System.out.println(args[0]);
    System.out.println(args[1]);
    System.out.println(args[2]);
  }
} ///:~
```

The **System.exit( )** call terminates the program and passes its argument back to the operating system as a result (with most operating systems, a non-zero result indicates that the program execution failed).

Exercise 8

```
//: c02:E08_AllTheColorsOfTheRainbow.java
/****************** Exercise 8 *****************
 * Turn the AllTheColorsOfTheRainbow example into
 * a program that compiles and runs.
 ***********************************************/
public class E08_AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void changeTheHueOfTheColor(int newHue) {
    anIntegerRepresentingColors = newHue;
  }
  public static void main(String[] args) {
    E08_AllTheColorsOfTheRainbow all =
      new E08_AllTheColorsOfTheRainbow();
    all.changeTheHueOfTheColor(27);
  }
} ///:~
```

Exercise 9

```
//: c02:E09_HelloDate.java
// {Javadoc: [HelloDate]}
/****************** Exercise 9 *****************
 * Find the code for the second version of
 * HelloDate.java, which is the simple comment
 * documentation example. Execute javadoc on the
 * file and view the results with your Web
 * browser.
 ***********************************************/
import java.util.*;

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 4.0
*/
public class E09_HelloDate {
  /** Sole entry point to class & application
   * @param args array of string arguments
   * @return No return value
   * @exception exceptions No exceptions thrown
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
} ///:~
```

Look at the **build.xml** file for this chapter (in the directory c02) to see the Ant directives for creating a directory and generating the resulting javadoc documentation in that directory. Note that **javadoc** doesn't automatically create the destination directory.

Exercise 10

```
//: c02:E10_DocTest.java
// {Javadoc: [DocTest]}
/****************** Exercise 10 *****************
 * Turn docTest into a file that compiles and
 * then run it through javadoc. Verify the
 * resulting documentation with your Web browser.
 ***********************************************/
/** A class comment */
```

```
public class E10_DocTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
} ///:~
```

You must still execute **javadoc** and verify the results, but you can see the commands that are added to the *Ant* build scripts, as the
{Javadoc: [DocTest]}
comment directive. This creates a directory named **DocTest** and runs **javadoc** to produce the documentation in this new directory.

Exercise 11

```
//: c02:E11_DocTest2.java
// {Javadoc: [DocTest2]}
/****************** Exercise 11 *****************
 * Add an HTML list of items to the documentation
 * in Exercise 10.
 ***********************************************/
/** A class comment
* <pre>
* System.out.println(new Date());
* </pre>
*/
public class E11_DocTest2 {
  /** A variable comment */
  public int i;
  /** A method comment
  * You can <em>even</em> insert a list:
  * <ol>
  * <li> Item one
  * <li> Item two
  * <li> Item three
  * </ol>
  */
  public void f() {}
} ///:~
```

I simply added the HTML code fragments from the chapter examples.

You must still execute **javadoc** and verify the results, but you can see the commands that are added to the *Ant* build file to do this, as the **{Javadoc: [DocTest2]}** comment directive.

Exercise 12

```
//: c02:E12_HelloWorldDoc.java
// {Javadoc: [HelloWorldDoc]}
/****************** Exercise 12 *****************
 * Take the program in Exercise 1 and add comment
 * documentation to it. Extract this comment
 * documentation into an HTML file using javadoc
 * and view it with your Web browser.
 ***********************************************/
/** A first example from <I>Thinking in Java,
* 3rd edition</I>. Demonstrates the basic class
* structure and the creation of a
* <code>main()</code> method.
*/
public class E12_HelloWorldDoc {
  /** The <code>main()</code> method which is
  called when the program is executed by saying
  <code>java E12_HelloWorldDoc</code>.
  @param args array passed from the command-line
  */
  public static void main(String args[]) {
    System.out.println("Hello, world!");
  }
} ///:~
```

You must still execute **javadoc** and verify the results, but you can see the commands that are added to the *Ant* build file to do this, as the **{Javadoc: [HelloWorldDoc]}** comment directive.

Exercise 13

```
//: c02:E13_OverloadingDoc.java
// {Javadoc: [OverloadingDoc]}
/****************** Exercise 13 *****************
 * In Chapter 4, locate the Overloading.java
 * example and add javadoc documentation. Extract
 * this comment documentation into an HTML file
 * using javadoc and view it with your Web browser.
 ***********************************************/

/** Model of a single arboreal unit. */
class Tree {
  /** Current vertical aspect to the tip. */
  int height;
  /** Plant a seedling. Assume height can
      be considered as zero. */
  Tree() {
    System.out.println("Planting a seedling");
    height = 0;
  }
  /** Transplant an existing tree with a given height. */
  Tree(int i) {
    System.out.println("Creating new Tree that is "
      + i + " feet tall");
    height = i;
  }
  /** Produce information about this unit. */
  void info() {
    System.out.println("Tree is " + height + " feet tall");
  }
  /** Produce information with optional message. */
  void info(String s) {
    System.out.println(s + ": Tree is "
      + height + " feet tall");
  }
}

/** Simple test code for Tree class */
public class E13_OverloadingDoc {
  /** Creates <b>Tree</b> objects and exercises the two
      different <code>info()</code> methods. */
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
} ///:~
```

Note the use of HTML inside the Javadoc for **main( )**.

Chapter 3

Exercise 1

```
//: c03:E01_Precedence.java
/****************** Exercise 1 *****************
 * There are two expressions in the section
 * labeled "precedence" early in this chapter.
 * Put these expressions into a program and
 * demonstrate that they produce different
 * results.
 ***********************************************/
public class E01_Precedence {
  static int a,
    x = 40,
    y = 60,
    z = 10;
  public static void main(String[] args) {
    a = x + y - 2/2 + z;
    System.out.println(a);
    a = x + (y - 2)/(2 + z);
    System.out.println(a);
  }
} ///:~
```

The output is 109 & 44. The difference is because the default order of evaluation is changed by the use of the parentheses.

Exercise 2

```
//: c03:E02_Ternary.java
/****************** Exercise 2 *****************
 * Put the methods ternary() and alternative()
 * into a working program.
 ***********************************************/
public class E02_Ternary {
  static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
  }
  static int alternative(int i) {
    if(i < 10)
      return i * 100;
    else
      return i * 10;
  }
  public static void main(String[] args) {
    System.out.println(ternary(9));
    System.out.println(ternary(11));
    System.out.println(alternative(9));
    System.out.println(alternative(11));
  }
} ///:~
```

Exercise 3

```
//: c03:E03_IfElse3.java
/****************** Exercise 3 ******************
 * From the sections labeled "if-else" and
 * "return", modify the two test() methods so
 * that testval is tested to see if it is within
 * the range between (and including) the
 * arguments begin and end.
 ***********************************************/
public class E03_IfElse3 {
  static boolean test(int testval, int begin, int end) {
    boolean result = false;
    if(testval >= begin && testval <= end)
      result = true;
    return result;
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5, 15));
    System.out.println(test(5, 10, 15));
    System.out.println(test(5, 5, 5));
  }
} ///:~
```

Since the **test( )** methods are now only testing for two conditions, I took the liberty of changing the return value to **boolean**.

By using **return** in the following program, notice that no intermediate **result** variable is necessary:

```
//: c03:E03_IfElse4.java
// No intermediate 'result' value necessary:
public class E03_IfElse4 {
  static boolean test(int testval, int begin, int end) {
    if(testval >= begin && testval <= end)
      return true;
    return false;
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5, 15));
    System.out.println(test(5, 10, 15));
    System.out.println(test(5, 5, 5));
  }
} ///:~
```

Exercise 4

```
//: c03:E04_To100.java
/****************** Exercise 4 ******************
 * Write a program that prints values from one to
 * 100.
 ***********************************************/
public class E04_To100 {
  public static void main(String[] args) {
    for(int i = 1; i <= 100; i++)
      System.out.println(i);
  }
} ///:~
```

This is the most trivial use of a **for** loop.

Exercise 5

```
//: c03:E05_Break47.java
```

```
/****************** Exercise 5 ******************
 * Modify Exercise 4 so that the program exits by
 * using the break keyword at value 47. Try using
 * return instead.
 ***********************************************/
public class E05_Break47 {
  public static void main(String[] args) {
    for(int i = 1; i <= 100; i++) {
      System.out.println(i);
      // if(i == 47) break;
      if(i == 47) return;
    }
  }
} ///:~
```

The above example includes a commented line of code to use **break**, as well. The **break** will exit the **for** loop, which puts it at the end of **main( )**, whereas **return** exits directly from the current method, which happens to be **main( )**.

Exercise 6

```
//: c03:E06_CompareStrings.java
/****************** Exercise 6 ******************
 * Write a method that takes two String
 * arguments, and uses all the boolean
 * comparisons to compare the two Strings and
 * print the results. For the == and !=, also
 * perform the equals() test. In main(), call
 * your method with some different String
 * objects.
 ***********************************************/
public class E06_CompareStrings {
  public static void p(String s, boolean b) {
    System.out.println(s + ": " + b);
  }
  public static void compare(String lval, String rval) {
    System.out.println("lval: " + lval + " rval: " + rval);
    //! p("lval < rval: " + lval < rval);
    //! p("lval > rval: " + lval > rval);
    //! p("lval <= rval: " + lval <= rval);
    //! p("lval >= rval: " + lval >= rval);
    p("lval == rval", lval == rval);
    p("lval != rval", lval != rval);
    p("lval.equals(rval)", lval.equals(rval));
  }
  public static void main(String[] args) {
    compare("Hello", "Hello");
    String s = new String("Hello");
    compare("Hello", s);
    compare("Hello", "Goodbye");
  }
} ///:~
```

This is a bit of a trick question, because the only comparisons that actually compile are **==** and **!=**. But it also points out the important difference between the **==** and **!=** operators, which compare references, and **equals( )**, which actually compares content.

The output of this program is:

```
lval: Hello rval: Hello
lval == rval: true
lval != rval: false
lval.equals(rval): true
lval: Hello rval: Hello
lval == rval: false
lval != rval: true
lval.equals(rval): true
lval: Hello rval: Goodbye
lval == rval: false
lval != rval: true
lval.equals(rval): false
```

Remember that quoted character arrays also produce references to **String** objects. In the first case, the compiler is smart enough to recognize that the two strings actually contain the same values. Because **String** objects are immutable – you cannot change their contents – the compile can merge the two **String** objects into one. This is why **==** returns **true** in that case. However, when a separate **String s** is created, even though it has the same contents, a distinct object is created and therefore the **==** returns **false**. The only reliable way to compare objects for equality is with **equals( )**; be wary if you see a comparison using **==**, which always compares to see if two references are identical (that is, whether they point to the same object).

Exercise 7

```
//: c03:E07_RandomInts.java
/****************** Exercise 7 ******************
 * Write a program that generates 25 random int
 * values. For each value, use an if-else
 * statement to classify it as greater than, less
 * than or equal to a second randomly-generated
 * value.
 ***********************************************/
import java.util.*;

public class E07_RandomInts {
  static Random r = new Random();
  public static void compareRand() {
    int a = r.nextInt();
    int b = r.nextInt();
    System.out.println("a = " + a + ", b = " + b);
    if(a < b)
      System.out.println("a < b");
    else if(a > b)
      System.out.println("a > b");
    else
      System.out.println("a = b");
  }
  public static void main(String[] args) {
    for(int i = 0; i < 25; i++)
      compareRand();
  }
} ///:~
```

Part of this is an exercise in Standard Java Library usage. If you haven't become familiar with the HTML documentation for the JDK, downloadable from *java.sun.com*, do so now. If you go to the index, and select “R” you will quickly come across the various choices for generating random numbers.

In the solution above, I created a method that generates the random numbers and compares them, and then I just call that method 25 times. In your solution, you may have created all the code inline, inside **main( )**.

Exercise 8

```
//: c03:E08_RandomInts2.java
// {RunByHand}
/****************** Exercise 8 ******************
 * Modify Exercise 7 so that your code is
 * surrounded by an "infinite" while loop. It
 * will then run until you interrupt it from the
 * keyboard (typically by pressing Control-C).
 ***********************************************/
import java.util.*;

public class E08_RandomInts2 {
  static Random r = new Random();
  public static void compareRand() {
    int a = r.nextInt();
    int b = r.nextInt();
    System.out.println("a = " + a + ", b = " + b);
    if(a < b)
      System.out.println("a < b");
    else if(a > b)
      System.out.println("a > b");
    else
      System.out.println("a = b");
  }
  public static void main(String[] args) {
    while(true)
      compareRand();
  }
} ///:~
```

This requires only a minor change, in **main( )**.

Exercise 9

```
//: c03:E09_FindPrimes.java
/****************** Exercise 9 ******************
 * Write a program that uses two nested for loops
 * and the modulus operator (%) to detect and
 * print prime numbers (integral numbers that are
 * not evenly divisible by any other numbers
 * except for themselves and 1).
 ***********************************************/
public class E09_FindPrimes {
  public static void main(String[] args) {
    int max = 100;
    // Get the max value from the command line,
    // if the argument has been provided:
    if(args.length != 0)
      max = Integer.parseInt(args[0]);
    for(int i = 1; i < max; i++) {
      boolean prime = true;
      for(int j = 2; j < i; j++)
        if(i % j == 0) prime = false;
      if(prime)
        System.out.println(i);
    }
  }
} ///:~
```

I've thrown in a bit extra here, which is the possibility of taking the **max** value from the command line. If you want to get an argument from the command line there are a few things you need to know. First, **arg[0]** is not the name of the program, as it is in C, but rather the first command-line argument. Second, arguments come in as a **String** array, so you must perform the conversion to whatever you actually need. If you want an **int**, the method is not particularly obvious: it's a **static** method of class **Integer** called **parseInt( )**. If you forget it, it can be a bit difficult to locate using the HTML documentation because you have to remember "parse" in order to find it, or that it's part of class **Integer**.

Exercise 10

This is basically just a straightforward exercise of all the behaviors of **switch**. I've made two separate programs, one with the **break**s and one without. Here's the version with **break**s:

```
//: c03:E10_SwitchDemo.java
/****************** Exercise 10 *****************
 * Create a switch statement that prints a
 * message for each case, and put the switch
 * inside a for loop that tries each case. Put a
 * break after each case and test it, then remove
 * the breaks and see what happens.
 ***********************************************/
public class E10_SwitchDemo {
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++)
      switch(i) {
        case 1: System.out.println("case 1");
                break;
        case 2: System.out.println("case 2");
                break;
        case 3: System.out.println("case 3");
                break;
        case 4: System.out.println("case 4");
                break;
        case 5: System.out.println("case 5");
                break;
        default: System.out.println("default");
      }
  }
} ///:~
```

The value of **i** is intentionally ranged out of bounds of the cases, to see what happens. Here's the output:

```
default
case 1
case 2
case 3
case 4
case 5
default
```

You can see that anything that doesn't match one of the cases goes to the **default** statement.

Here's the same program with the **break**s removed:

```
//: c03:E10_SwitchDemo2.java
// E10_SwitchDemo.java with the breaks removed.

public class E10_SwitchDemo2 {
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++)
      switch(i) {
        case 1: System.out.println("case 1");
        case 2: System.out.println("case 2");
        case 3: System.out.println("case 3");
        case 4: System.out.println("case 4");
        case 5: System.out.println("case 5");
        default: System.out.println("default");
      }
  }
} ///:~
```

Now the output is:

```
default
case 1
case 2
case 3
case 4
case 5
default
case 2
case 3
case 4
case 5
default
case 3
case 4
case 5
```

```
default
case 4
case 5
default
case 5
default
default
```

Without the **break**, each case falls through to the next one. So when you end up in **case 1**, you get all the other cases as well. Thus, you’ll almost always want a **break** at the end of each **case**.

Additional Exercise:

```
//: c03:EXTRA1_Fibonacci.java
// {Args: 20}
/****************** Exercise EXTRA2 *********************
 * A Fibonacci Sequence is the sequence of numbers
 * 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on, where each
 * number (from the third on) is the sum of the previous
 * two. Create a method that takes an integer as an
 * argument and prints out that many Fibonacci numbers
 * starting from the beginning. E.g.: if you run
 * java Fibonacci 5
 * (where Fibonacci is the name of the class) the output
 * will be: 1, 1, 2, 3, 5.
 ******************************************************/
public class EXTRA1_Fibonacci {
  static int fib(int n) {
    if (n <= 2)
      return 1;
    else
      return fib(n-1) + fib(n-2);
  }
  public static void main(String[] args) {
    // Get the max value from the command line:
    int n = Integer.parseInt(args[0]);
    if(n < 0) {
      System.out.println("Cannot use negative numbers");
      return;
    }
    for(int i = 1; i <= n; i++)
      System.out.print(fib(i) + ", ");
  }
} ///:~
```

This is a commonly-solved problem in introductory programming classes, so presumably you looked up the typical algorithm on the internet. This uses *recursion*, which means that a function calls itself until it reaches a bottoming-out condition and returns.

Additional Exercise:

(From Dan Forhan) Write a Java program that will find all the possible 4-digit "vampire numbers." A vampire number is defined as a four digit number in which the two numbers multiplied are in the product, for example, 15 x 93 = 1395. The program should produce no duplicates.

```
//: c03:EXTRA2_Vampire.java

public class EXTRA2_Vampire {
  public static void main(String[] args) {
   int[] startDigit = new int[4];
   int[] productDigit = new int[4];
    for(int num1 = 10; num1 <= 99; num1++)
      for(int num2 = 10; num2 <= 99; num2++) {
        int product = num1 * num2;
        startDigit[0] = num1 / 10;
        startDigit[1] = num1 % 10;
        startDigit[2] = num2 / 10;
        startDigit[3] = num2 % 10;
        productDigit[0] = product / 1000;
        productDigit[1] = (product % 1000) / 100;
        productDigit[2] = product % 1000 % 100 / 10;
        productDigit[3] = product % 1000 % 100 % 10;
        int count = 0;
        for(int x = 0; x < 4; x++)
          for(int y = 0; y < 4; y++) {
            if(productDigit[x] == startDigit[y]) {
              count++;
              productDigit[x] = -1;
              startDigit[y] = -2;
              if(count == 4)
                System.out.println(num1 + " * " + num2
                  + " : " + product);
            }
          }
      }
  }
} ///:~
```

Chapter 4

Exercise 1

```
//: c04:E01_DefaultConstructor.java
/****************** Exercise 1 ******************
 * Create a class with a default constructor (one
 * that takes no arguments) that prints a
 * message. Create an object of this class.
 ***********************************************/
public class E01_DefaultConstructor {
  E01_DefaultConstructor() {
    System.out.println("Default constructor");
  }
  public static void main(String args[]) {
    new E01_DefaultConstructor();
  }
} ///:~
```

Since we are only creating the **DefaultConstructor( )** object for the "side effect" of its construction process, there is no need to create and hold a reference to the object.

Exercise 2

```
//: c04:E02_OverloadedConstructor.java
/****************** Exercise 2 *****************
 * Add an overloaded constructor to Exercise 1
 * that takes a String argument and prints it
 * along with your message.
 ***********************************************/

public class E02_OverloadedConstructor {
  E02_OverloadedConstructor() {
    System.out.println("Default constructor");
  }
  E02_OverloadedConstructor(String s) {
    System.out.println("String arg constructor");
    System.out.println(s);
  }
  public static void main(String args[]) {
    // Call default version:
    new E02_OverloadedConstructor();
    // Call overloaded version:
    new E02_OverloadedConstructor("Overloaded");
  }
} ///:~
```

A second way to do this is to explicitly call the default constructor inside the overloaded constructor using **this**:

```
//: c04:E02_OverloadedConstructor2.java

public class E02_OverloadedConstructor2 {
  E02_OverloadedConstructor2() {
    System.out.println("Default constructor");
  }
  E02_OverloadedConstructor2(String s) {
    this();
    System.out.println(s);
  }
  public static void main(String args[]) {
    // Call default version:
    new E02_OverloadedConstructor2();
    // Call overloaded version:
    new E02_OverloadedConstructor2("Overloaded");
  }
} ///:~
```

Exercise 3

```
//: c04:E03_ObjectReferences.java
/****************** Exercise 3 *****************
 * Create an array of object references of the
 * class you created in Exercise 2, but don't
 * actually create objects to assign into the
 * array. When you run the program, notice
 * whether the initialization messages from the
 * constructor calls are printed.
 ***********************************************/
public class E03_ObjectReferences {
  // You can define the array as a field
  // in the class ...
  E02_OverloadedConstructor[] array1 =
    new E02_OverloadedConstructor[5];
  public static void main(String args[]) {
    // Or as a temporary inside main:
    E02_OverloadedConstructor[] array2 =
      new E02_OverloadedConstructor[5];
  }
} ///:~
```

The above code only creates the array, not the objects that go into the array. Therefore, you won't see the initialization messages in **E02_OverloadedConstructor**'s constructors since no instances of that class are ever created.

Exercise 4

```
//: c04:E04_ObjectArray.java
/****************** Exercise 4 *****************
 * Complete Exercise 3 by creating objects to
 * attach to the array of references.
 ***********************************************/
public class E04_ObjectArray {
  E02_OverloadedConstructor[] array =
    new E02_OverloadedConstructor[5];
  public E04_ObjectArray() {
    for(int i = 0; i < array.length; i++)
      array[i] = new E02_OverloadedConstructor();
  }
  // An overloaded constructor that calls the
  // overloaded constructor from Exercise 2:
  public E04_ObjectArray(String s) {
    for(int i = 0; i < array.length; i++)
      array[i] = new E02_OverloadedConstructor(s);
  }
  public static void main(String args[]) {
    new E04_ObjectArray();
    new E04_ObjectArray("Overloaded");
  }
} ///:~
```

Since the array is defined as a field of the class, it must be initialized in the constructor (if you try to initialize it inside **main( )** you'll get a complaint that a non-**static** field cannot be accessed inside a **static** method). I've taken an extra step here and created a second, overloaded constructor that initializes the array by calling the overloaded constructor from Exercise 2. The output is:

```
Default constructor
Default constructor
Default constructor
Default constructor
Default constructor
String arg constructor
Overloaded
String arg constructor
Overloaded
String arg constructor
Overloaded
String arg constructor
Overloaded
String arg constructor
Overloaded
```

Exercise 5

```
//: c04:E05_StringArray.java
/****************** Exercise 5 ******************
 * Create an array of String objects and assign a
 * string to each element. Print the array using
 * a for loop.
 ***********************************************/
public class E05_StringArray {
  public static void main(String args[]) {
    // Doing it the hard way:
    String sa1[] = new String[4];
    sa1[0] = "These";
    sa1[1] = "are";
    sa1[2] = "some";
    sa1[3] = "strings";
    for(int i = 0; i < sa1.length; i++)
      System.out.println(sa1[i]);
    // Using aggregate initialization to
    // make it easier:
    String sa2[] = {
      "These", "are", "some", "strings"
    };
    for(int i = 0; i < sa2.length; i++)
      System.out.println(sa2[i]);
  }
} ///:~
```

The above solution shows both ways you can do it: explicitly creating the array object and assigning a string into each slot by hand, or using *aggregate initialization*, which creates the array object and does the initialization for you.

Exercise 6

```
//: c04:E06_OverloadedDog.java
/****************** Exercise 6 *****************
 * Create a class called Dog with an overloaded
 * bark() method. This method should be
 * overloaded based on various primitive data
 * types, and print different types of barking,
 * howling, etc., depending on which overloaded
 * version is called. Write a main() that calls
 * all the different versions.
 ***********************************************/
class Dog {
  public void bark() {
    System.out.println("Default bark!");
  }
  public void bark(int i) {
    System.out.println("int bark = howl");
  }
  public void bark(double f) {
    System.out.println("float bark = yip");
  }
  // Etc. ...
}

public class E06_OverloadedDog {
  public static void main(String args[]) {
    Dog dog = new Dog();
    dog.bark();
    dog.bark(1);
    dog.bark(1.1);
  }
} ///:~
```

Additional Exercise: (This is a trick question, so watch out). Write a class with a method **boolean print(int)** that prints a value and returns a **boolean**. Now overload the method to return a **long**. (Note: this is similar to some kinds of questions on the Sun Java Certification Exam).

Exercise 7

```
//: c04:E07_SwappedArguments.java
/****************** Exercise 7 ******************
 * Modify Exercise 6 so that two of the
 * overloaded methods have two arguments (of two
 * different types), but in reversed order
 * relative to each other. Verify that this
 * works.
 ***********************************************/
class Dog2 {
  public void bark(int i, double f) {
    System.out.println("int, double bark");
  }
  public void bark(double f, int i) {
    System.out.println("double, int bark");
  }
}

public class E07_SwappedArguments {
  public static void main(String args[]) {
    Dog2 dog = new Dog2();
    dog.bark(1, 2.2);
    dog.bark(2.2, 1);
  }
} ///:~
```

This exercise emphasizes that it's not only the type of arguments that are used to distinguish an overloaded method, but also the order of the arguments. In the above example, the two versions of **bark( )** are unique.

Exercise 8

```
//: c04:E08_SynthesizedConstructor.java
/****************** Exercise 8 ******************
 * Create a class without a constructor, and then
 * create an object of that class in main() to
 * verify that the default constructor is
 * automatically synthesized.
 ***********************************************/
public class E08_SynthesizedConstructor {
  public static void main(String args[]) {
    // Call the synthesized default constructor
    // for this class:
    new E08_SynthesizedConstructor();
  }
} ///:~
```

Since the constructor can be called, it must have been created, even if you can't see it.

Exercise 9

```
//: c04:E09_ThisMethodCall.java
/****************** Exercise 9 *****************
 * Create a class with two methods. Within the
 * first method, call the second method twice:
 * the first time without using this, and the
 * second time using this.
 ***********************************************/
```

```
public class E09_ThisMethodCall {
  public void a() {
    b();
    this.b();
  }
  public void b() {
    System.out.println("b() called");
  }
  public static void main(String args[]) {
    new E09_ThisMethodCall().a();
  }
} ///:~
```

This exercise just shows how **this** refers to the current object. I certainly don't recommend the **this.b( )** form of method call – only use **this** when necessary.

Exercise 10

```
//: c04:E10_ThisConstructorCall.java
/****************** Exercise 10 ****************
 * Create a class with two (overloaded)
 * constructors. Using this, call the second
 * constructor inside the first one.
 ***********************************************/
public class E10_ThisConstructorCall {
  public E10_ThisConstructorCall(String s) {
    System.out.println("s = " + s);
  }
  public E10_ThisConstructorCall(int i) {
    this("i = " + i);
  }
  public static void main(String args[]) {
    new E10_ThisConstructorCall("String call");
    new E10_ThisConstructorCall(47);
  }
} ///:~
```

Exercise 11

```
//: c04:E11_FinalizeCall.java
/****************** Exercise 11 *****************
 * Create a class with a finalize() method that
 * prints a message. In main(), create an object
 * of your class. Explain the behavior of your
 * program.
 ***********************************************/
public class E11_FinalizeCall {
  protected void finalize() {
    System.out.println("Finalize called");
  }
  public static void main(String args[]) {
    new E11_FinalizeCall();
  }
} ///:~
```

You probably won't see the finalizer called. This is typically because the program hasn't generated enough garbage for the collector to be run.

Exercise 12

```
//: c04:E12_FinalizeAlwaysCalled.java
/****************** Exercise 12 ****************
 * Modify Exercise 11 so that your finalize()
 * will always be called.
 ***********************************************/
public class E12_FinalizeAlwaysCalled {
  protected void finalize() {
    System.out.println("Finalize called");
  }
  public static void main(String args[]) {
    new E12_FinalizeAlwaysCalled();
    System.gc();
  }
} ///:~
```

Calling **System.gc( )** will *probably* cause your finalizer to be called. I say "probably" because the behavior of finalize has been uncertain from one version of the JDK to another. In addition, calling **System.gc( )** is just a request, and it doesn't guarantee that the garbage collector will actually be run.

Another solution that was put into one version of the JDK, and then deprecated in the next version (just to assure you that they know what they're doing there at Sun) is to call **System.runFinalizersOnExit( )**. Since this is deprecated, you can't really use it anymore. Basically, there's no way to ensure that **finalize( )** will be called.

Exercise 13

```
//: c04:E13_TankWithTerminationCondition.java
/****************** Exercise 13 *****************
 * Create a class called Tank that can be filled
 * and emptied, and has a termination condition that it
 * must be empty when the object is cleaned up.
 * Write a finalize() that verifies this termination
 * condition. In main(), test the possible
 * scenarios that can occur when your Tank is
 * used.
 ***********************************************/
class Tank {
  static int counter = 0;
  int id = counter++;
  boolean full = false;
  public Tank() {
    System.out.println("Tank " + id + " created");
    full = true;
  }
  public void empty() {
    full = false;
  }
  protected void finalize() {
    if(full)
      System.out.println(
        "Error: tank " + id + " must be empty at cleanup");
    else
      System.out.println("Tank " + id + " cleaned up OK");
  }
  public String toString() {
    return "Tank " + id;
  }
}

public class E13_TankWithTerminationCondition {
  public static void main(String args[]) {
    new Tank().empty();
    new Tank();
    // Don't empty the second one
    System.gc(); // Force finalization?
  }
} ///:~
```

Note that I intentionally did not create references to the two instances of type **Tank**, because then those references would be in scope when **System.gc( )** was called and they wouldn't be cleaned up and thus they wouldn't be finalized (another option is to explicitly set references to zero when you want them to be garbage collected – try it by modifying the above example).

The output is:

```
Tank 0 created
Tank 1 created
Tank 0 cleaned up OK
Error: tank 1 must be empty at cleanup
```

Because you can't be sure finalizers will be called, they have only a limited use and this is one of them – checking the state of objects when they *do* get run, to ensure they've been cleaned up properly.

Exercise 14

```
//: c04:E14_DefaultInitialization.java
/****************** Exercise 14 ****************
 * Create a class containing an int and a char
 * that are not initialized, and print their
 * values to verify that Java performs default
 * initialization.
 ***********************************************/
```

```
public class E14_DefaultInitialization {
  int i;
  char c;
  public E14_DefaultInitialization() {
    System.out.println("i = " + i);
    System.out.println("c = " + c);
  }
  public static void main(String args[]) {
    new E14_DefaultInitialization();
  }
} ///:~
```

When you run the program you'll see that both variables are given default values: 0 for the int, and a space for the char.

Exercise 15

```
//: c04:E15_StringRefInitialization.java
/****************** Exercise 15 ****************
 * Create a class containing an uninitialized
 * String reference. Demonstrate that this
 * reference is initialized by Java to null.
 ***********************************************/
public class E15_StringRefInitialization {
  String s;
  public static void main(String args[]) {
    E15_StringRefInitialization sri =
      new E15_StringRefInitialization();
    System.out.println("sri.s = " + sri.s);
  }
} ///:~
```

The output is:

```
sri.s = null
```

Exercise 16

```
//: c04:E16_StringInitialization.java
/****************** Exercise 16 ****************
```

```
 * Create a class with a String field that is
 * initialized at the point of definition, and
 * another one that is initialized by the
 * constructor. What is the difference between
 * the two approaches?
 ***********************************************/
public class E16_StringInitialization {
  String s1 = "Initialized at definition";
  String s2;
  public E16_StringInitialization(String s2i) {
    s2 = s2i;
  }
  public static void main(String args[]) {
    E16_StringInitialization si =
      new E16_StringInitialization(
        "Initialized at construction");
    System.out.println("si.s1 = " + si.s1);
    System.out.println("si.s2 = " + si.s2);
  }
} ///:~
```

The **s1** field is initialized before the constructor is entered. Technically, so is the **s2** field, as it is set to **null** as the object is created. The **s2** field has flexibility, since you can choose what value to give it as you call the constructor, whereas **s1** will always have the same value.

Exercise 17

```
//: c04:E17_StaticStringInitialization.java
/****************** Exercise 17 *****************
 * Create a class with a static String field that
 * is initialized at the point of definition, and
 * another one that is initialized by the static
 * block. Add a static method that prints both
 * fields and demonstrates that they are both
 * initialized before they are used.
 ***********************************************/
public class E17_StaticStringInitialization {
  static String s1 = "Initialized at definition";
  static String s2;
  static {
    s2 = "Initialized in static block";
  }
  public static void main(String args[]) {
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
  }
} ///:~
```

main( ) is a **static** method, so I just used that to print out the values.

Exercise 18

```
//: c04:E18_StringInstanceInitialization.java
/****************** Exercise 18 ****************
 * Create a class with a String that is
 * initialized using "instance initialization."
 * Describe a use for this feature (other than
 * the one specified in this book).
 ***********************************************/
public class E18_StringInstanceInitialization {
  String s;
  {
    s = "'instance initialization'";
  }
  public E18_StringInstanceInitialization() {
    System.out.println("Default constructor; s = " + s);
  }
  public E18_StringInstanceInitialization(int i) {
    System.out.println("int constructor; s = " + s);
  }
  public static void main(String args[]) {
    new E18_StringInstanceInitialization();
    new E18_StringInstanceInitialization(1);
  }
} ///:~
```

The book states that the reason for instance initialization is to provide basic constructor activites for anonymous inner classes (which you haven't learned about yet), because these cannot have constructor methods (constructors require names, and an anonymous inner class is anonymous – it has no name).

The reason that you *might* justify the use of instance initialization in an ordinary class is to perform operations that will always need to be performed regardless of which constructor gets called, especially those operations that are too complex to perform during initialization at the point of definition (an expression involving a **for** loop, for example, or one that involves opening and reading a file). It is not that common find a situation where common construction code can be factored into instance initialization, but it does happen. However, you can also put the basic operations in a named constructor and call that constructor using the **this** syntax for constructor calls. In fact, you should be somewhat suspicious of code that uses instance initialization in anything but an anonymous class.

Note that in the above example there are two constructors, and when you run the program you'll see that instance initialization occurs before either constructor runs.

Exercise 19

```
//: c04:E19_TwoDDoubleArray.java
/****************** Exercise 19 *****************
 * Write a method that creates and initializes a
 * two-dimensional array of double. The size of
 * the array is determined by the arguments of
 * the method, and the initialization values are
 * a range determined by beginning and ending
 * values that are also arguments of the method.
 * Create a second method that will print the
 * array generated by the first method. In main()
 * test the methods by creating and printing
 * several different sizes of arrays.
 ***********************************************/
public class E19_TwoDDoubleArray {
  public static double[][] twoDDoubleArray(
    int xLen, int yLen, double valStart, double valEnd) {
    double[][] array = new double[xLen][yLen];
    double increment = (valEnd - valStart)/(xLen * yLen);
    double val = valStart;
    for(int i = 0; i < array.length; i++)
      for(int j = 0; j < array[i].length; j++) {
        array[i][j] = val;
        val += increment;
      }
    return array;
  }
  public static void printArray(double[][] array) {
    for(int i = 0; i < array.length; i++) {
      for(int j = 0; j < array[i].length; j++)
        System.out.print(" " + array[i][j]);
      System.out.println();
    }
  }
  public static void main(String args[]) {
    double[][] twoD = twoDDoubleArray(4, 6, 47.0, 99.0);
    printArray(twoD);
    System.out.println("**********************");
    double[][] twoD2 = twoDDoubleArray(2, 2, 47.0, 99.0);
    printArray(twoD2);
    System.out.println("**********************");
    double[][] twoD3 = twoDDoubleArray(9, 5, 47.0, 99.0);
    printArray(twoD3);
  }
} ///:~
```

The step value for the initialization range is calculated by dividing the range by the number of elements in the array.

Exercise 20

```
//: c04:E20_ThreeDDoubleArray.java
/****************** Exercise 20 ****************
 * Repeat Exercise 19 for a three-dimensional
 * array.
 ***********************************************/
public class E20_ThreeDDoubleArray {
  public static double[][][] threeDDoubleArray(
    int xLen, int yLen, int zLen,
    double valStart, double valEnd) {
    double[][][] array = new double[xLen][yLen][zLen];
    double increment =
      (valEnd - valStart)/(xLen * yLen * zLen);
    double val = valStart;
    for(int i = 0; i < array.length; i++)
      for(int j = 0; j < array[i].length; j++)
        for(int k = 0; k < array[i][j].length; k++) {
          array[i][j][k] = val;
          val += increment;
        }
    return array;
  }
  public static void printArray(double[][][] array) {
    for(int i = 0; i < array.length; i++) {
      for(int j = 0; j < array[i].length; j++) {
        for(int k = 0; k < array[i][j].length; k++)
          System.out.println(" " + array[i][j][k]);
        System.out.println();
      }
      System.out.println();
    }
  }
  public static void main(String args[]) {
    double[][][] threeD =
      threeDDoubleArray(4, 6, 2, 47.0, 99.0);
    printArray(threeD);
    System.out.println("**********************");
    double[][][] threeD2 =
      threeDDoubleArray(2, 2, 5, 47.0, 99.0);
    printArray(threeD2);
    System.out.println("**********************");
    double[][][] threeD3 =
      threeDDoubleArray(9, 5, 7, 47.0, 99.0);
    printArray(threeD3);
  }
} ///:~
```

As complex as this might seem, it's still vastly simpler than doing it in C or C++, and you get array bounds checking built in. In C/C++, if you ran off the end of a multi-dimensional array, there is a good chance you wouldn't catch the error.

Exercise 21

```
//: c04:E21_LeftToReader.java
/****************** Exercise 21 *****************
 * Comment the line marked (1) in
 * ExplicitStatic.java and verify that the static
 * initialization clause is not called. Now
 * uncomment one of the lines marked (2) and
 * verify that the static initialization clause
 * is called. Now uncomment the other line marked
 * (2) and verify that static initialization only
 * occurs once.
 ***********************************************/
public class E21_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Chapter 5

Note that many of these exercises simply take you through the steps of discovering issues with packages. This is why so many of them are left to the reader.

Exercise 1

```
//: c05:E01_ImplicitImports.java
/****************** Exercise 1 *****************
 * Write a program that creates an ArrayList
 * object without explicitly importing
 * java.util.*.
 ***********************************************/
public class E01_ImplicitImports {
  public static void main(String args[]) {
    java.util.ArrayList al = new java.util.ArrayList();
  }
} ///:~
```

To do this, any references to the object must be made using its full package name.

Exercise 2

```
//: c05:E02_LeftToReader.java
/****************** Exercise 2 ******************
 * In the section labeled "package: the library
 * unit," turn the code fragments concerning
 * mypackage into a compiling and running set of
 * Java files.
 ***********************************************/
public class E02_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Exercise 3

```
//: c05:E03_LeftToReader.java
/****************** Exercise 3 ******************
 * In the section labeled "Collisions," take the
 * code fragments and turn them into a program,
 * and verify that collisions do in fact occur.
 ***********************************************/
public class E03_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Exercise 4

```
//: c05:E04_LeftToReader.java
/****************** Exercise 4 ******************
 * Generalize the class P defined in this chapter
 * by adding all the overloaded versions of
 * rint() and rintln() necessary to handle all
 * the different basic Java types.
 ***********************************************/
public class E04_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Exercise 5

```
//: c05:E05_AccessControl.java
/****************** Exercise 5 ******************
 * Create a class with public, private,
 * protected, and package-access data members and
 * method members. Create an object of this class
 * and see what kind of compiler messages you get
 * when you try to access all the class members.
 * Be aware that classes in the same directory
 * are part of the "default" package.
 ***********************************************/
package c05;

public class E05_AccessControl {
  public int a;
  private int b;
  protected int c;
  int d; // Package access
  public void f1() {}
  private void f2() {}
  protected void f3() {}
  void f4() {} // Package access
  public static void main(String args[]) {
    E05_AccessControl test = new E05_AccessControl();
    // No problem accessing everything inside
    // of main() for this class, since main()
    // is a member and therefore has access:
    test.a = 1;
    test.b = 2;
    test.c = 3;
    test.d = 4;
    test.f1();
    test.f2();
    test.f3();
    test.f4();
  }
}  ///:~
```

You can see that **main( )** has access to everything because it's a member. If you create a separate class within the same package, that class cannot access the private members:

```
//: c05:E05_Other.java
// A separate class in the same package cannot
// access private elements:
package c05;

public class E05_Other {
  public E05_Other() {
    E05_AccessControl test = new E05_AccessControl();
    test.a = 1;
    //! test.b = 2; // Can't access:  private
    test.c = 3;
    test.d = 4;
    test.f1();
    //! test.f2(); // Can't access:  private
    test.f3();
    test.f4();
  }
} ///:~
```

If you create a class in a separate package (which you can do either by explicitly using a package statement, or by simply putting it in a different directory) then it is unable to access anything but **public** members:

```
//: c05:other2:E05_Other2.java
// A separate class in the same package cannot
// access private elements:
package c05.other2;

public class E05_Other2 {
  public E05_Other2() {
    c05.E05_AccessControl test =
      new c05.E05_AccessControl();
    test.a = 1;
    //! test.b = 2; // Can't access: private
    //! test.c = 3; // Can't access: protected
    //! test.d = 4; // Can't access: package
    test.f1();
    //! test.f2(); // Can't access:  private
    //! test.f3(); // Can't access: protected
    //! test.f4(); // Can't access: package
  }
} ///:~
```

Exercise 6

```
//: c05:E06_ProtectedManipulation.java
/****************** Exercise 6 *****************
 * Create a class with protected data. Create a
 * second class in the same file with a method
 * that manipulates the protected data in the
 * first class.
 ***********************************************/
class WithProtected {
  protected int i;
}

public class E06_ProtectedManipulation {
  public static void main(String args[]) {
    WithProtected wp  = new WithProtected();
    wp.i = 47;
    System.out.println("wp.i = " + wp.i);
  }
} ///:~
```

The point of this exercise is to show that **protected** also means "package access" (aka "friendly"). That is, you can access **protected** fields within the same package. Just to be sure, try adding a **protected** method to **WithProtected** and accessing it from within **E06_ProtectedManipulation**.

Exercise 7

```
//: c05:E07_LeftToReader.java
/****************** Exercise 7 *****************
 * Change the class Cookie as specified in the section
 * labeled "protected:inheritance access" Verify that
 * bite() is not public.
 ***********************************************/
public class E07_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Just follow the instructions in the book.

Exercise 8

```
//: c05:E08_Widget.java
/****************** Exercise 8 *****************
 * In the section titled "Class access" you'll
 * find code fragments describing mylib and
 * Widget. Create this library, then create a
 * Widget in a class that is not part of the
 * mylib package.
 ***********************************************/
import c05.mylib.Widget;

public class E08_Widget {
  public static void main(String args[]) {
    new Widget();
  }
} ///:~
```

```
//: c05:mylib:Widget.java
package c05.mylib;

public class Widget {
  public Widget() {
    System.out.println("Making a Widget");
  }
} ///:~
```

Exercise 9

```
//: c05:E09_LeftToReader.java
/****************** Exercise 9 ******************
 * Create a new directory and edit your CLASSPATH
 * to include that new directory. Copy the
 * P.class file (produced by compiling
 * com.bruceeckel.tools.P.java) to your new
 * directory and then change the names of the
 * file, the P class inside, and the method names.
 * (You might also want to add additional output
 * to watch how it works.) Create another program
 * in a different directory that uses your new
 * class.
 *********************************************/
public class E09_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Exercise 10

```
//: c05:E10_ConnectionManager.java
/****************** Exercise 10 *****************
 * Following the form of the example Lunch.java,
 * create a class called ConnectionManager that
 * manages a fixed array of Connection objects.
 * The client programmer must not be able to
 * explicitly create Connection objects, but can
 * only get them via a static method in
 * ConnectionManager. When the ConnectionManager
 * runs out of objects, it returns a null
 * reference. Test the classes in main().
 *********************************************/
import c05.connection.*;

public class E10_ConnectionManager {
  public static void main(String args[]) {
    Connection c = ConnectionManager.getConnection();
    while(c != null) {
      System.out.println(c);
      c.doSomething();
      c = ConnectionManager.getConnection();
    }
  }
} ///:~
```

```
//: c05:connection:Connection.java
package c05.connection;

public class Connection {
  private static int counter = 0;
  private int id = counter++;
  Connection() {}
  public String toString() {
    return "Connection " + id;
  }
  public void doSomething() {}
} ///:~
```

```
//: c05:connection:ConnectionManager.java
package c05.connection;

public class ConnectionManager {
  private static Connection[] pool = new Connection[10];
  private static int counter = 0;
  static {
    for(int i = 0; i < pool.length; i++)
      pool[i] = new Connection();
  }
  // Very simple -- just hands out each one once:
  public static Connection getConnection() {
    if(counter < pool.length)
      return pool[counter++];
    return null;
  }
} ///:~
```

The **Connection** class uses a **static int** called **counter** to automatically give each **Connection** object an identifier, which it will produce as part of its **toString( )** representation. **Connection** also has a **doSomething( )** method, which is presumably the useful work that motivates you to use the **Connection** object.

Note that the constructor for **Connection** has package access, so it is unavailable outside of this package. The client programmer is therefore unable to access that constructor and cannot make instances of **Connection** directly. The only way to get **Connection** objects is through the **ConnectionManager**.

In the **ConnectionManager**, a **static** array of objects is initialized inside the *static clause*, which is only called once when the class is loaded (you can look up further details on static clauses in the book). In this version of the solution, the connection manager is trivial, since it only produces each **Connection** object in the array once.

You might want to create a slightly more sophisticated connection manager by allowing a connection to be checked back in when the client programmer is finished with it. Here's one way to accomplish this:

```
//: c05:E10_ConnectionManager2.java
// Connections that can be checked in.
import c05.connection2.*;

public class E10_ConnectionManager2 {
  public static void main(String args[]) {
    Connection[] ca = new Connection[10];
    // Use up all the connections
    for(int i = 0; i < 10; i++)
      ca[i] = ConnectionManager.getConnection();
    // Should produce "null" since there are no
    // more connections:
    System.out.println(ConnectionManager.getConnection());
    // Return connections, then get them out:
    for(int i = 0; i < 5; i++) {
      ca[i].checkIn();
      Connection c = ConnectionManager.getConnection();
      System.out.println(c);
      c.doSomething();
      c.checkIn();
    }
  }
} ///:~
```

```
//: c05:connection2:Connection.java
package c05.connection2;

public class Connection {
  private static int counter = 0;
  private int id = counter++;
  Connection() {}
  public String toString() {
    return "Connection " + id;
  }
  public void doSomething() {}
  public void checkIn() {
    ConnectionManager.checkIn(this);
  }
} ///:~
```

```
//: c05:connection2:ConnectionManager.java
package c05.connection2;

public class ConnectionManager {
  private static Connection[] pool = new Connection[10];
  static {
    for(int i = 0; i < pool.length; i++)
      pool[i] = new Connection();
  }
  // Produce the first available connection:
  public static Connection getConnection() {
    for(int i = 0; i < pool.length; i++)
      if(pool[i] != null) {
        Connection c = pool[i];
        pool[i] = null; // Indicates "in use"
        return c;
      }
    return null; // None left
  }
  public static void checkIn(Connection c) {
    for(int i = 0; i < pool.length; i++)
      if(pool[i] == null) {
        pool[i] = c; // Check it back in
        return;
      }
  }
} ///:~
```

When a **Connection** is checked out, its slot in **pool** is set to **null**. When the client programmer is done with the connection, **checkIn( )** will return it to the connection pool by finding a **null** slot in **pool** and assigning it there.

Of course, there are all kinds of problems with this approach. What if a client checks a **Connection** in and then continues to use it? What if they check it in more than once? The book *Thinking in Patterns* (available from www.BruceEckel.com) has a more thorough coverage of the problem of the connection pool.

Exercise 11

```
//: c05:E11_ForeignPackage.java

public class E11_ForeignPackage {
  public static void main(String args[]) {
    System.out.println(
      "See Solutions Guide document for details");
  }
} ///:~
```

Create the following file in the c05/local directory (which is presumably in your CLASSPATH):

```
//: c05:local:PackagedClass.java
package c05.local;

class PackagedClass {
  public PackagedClass() {
    System.out.println("Creating a packaged class");
  }
} ///:~
```

Then create the following file in a directory other than c05:

```
//: c05:foreign:Foreign.java
// {CompileByHand} to see results
package c05.foreign;
import c05.local.*;

public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
} ///:~
```

Explain why the compiler generates an error. Would making the **Foreign** class part of the **c05.local** package change anything?

Solution: PackagedClass is in its own package, and it is not a **public** class so it is unavailable outside of **package c05.local**. If **Foreign** was also part of **c05.local**, then it would have access to **PackagedClass**, since they would then both be in the same package.

Chapter 6

Exercise 1

```
//: c06:E01_SimpleInheritance.java
/****************** Exercise 1 *****************
 * Create two classes, A and B, with default
 * constructors (empty argument lists) that
 * announce themselves. Inherit a new class
 * called C from A, and create a member of class
 * B inside C. Do not create a constructor for C.
 * Create an object of class C and observe the
 * results.
 ***********************************************/
class A {
  public A() {
    System.out.println("A()");
  }
}

class B {
  public B() {
    System.out.println("B()");
  }
}

class C extends A {
  B b = new B();
}

public class E01_SimpleInheritance {
  public static void main(String args[]) {
    new C();
  }
} ///:~
```

The output is:

```
A()
B()
```

This shows that a constructor for **C** is automatically synthesized by the compiler, and that the base-class constructor is called first, then the member object constructors.

Exercise 2

```
//: c06:E02_SimpleInheritance2.java
/****************** Exercise 2 ******************
 * Modify Exercise 1 so that A and B have
 * constructors with arguments instead of default
 * constructors. Write a constructor for C and
 * perform all initialization within C's
 * constructor.
 ***********************************************/
class A2 {
  public A2(String s) {
    System.out.println("A2(): " + s);
  }
}

class B2 {
  public B2(String s) {
    System.out.println("2B(): " + s);
  }
}

class C2 extends A2 {
  B2 b;
  public C2(String s) {
    super(s);
    b = new B2(s);
  }
}

public class E02_SimpleInheritance2 {
  public static void main(String args[]) {
    new C2("Init string");
  }
} ///:~
```

(The ‘2’s were added to keep the class names from clashing, as they are in the same directory).

Remember that **super** calls the base-class constructor and must be the first call in a derived-class constructor. The output is:

```
A2(): Init string
B2(): Init string
```

Exercise 3

```
//: c06:E03_Composition.java
/****************** Exercise 3 *****************
 * Create a simple class. Inside a second class,
 * define a reference to an object of the first
 * class. Use lazy initialization to instantiate
 * this object.
 ***********************************************/
class Simple {
  String s;
  public Simple(String si) {
    s = si;
  }
  public String toString() {
    return s;
  }
  public void setString(String sNew) {
    s = sNew;
  }
}

class Second {
  Simple simple;
  String s;
  public Second(String si) {
    s = si; // 'simple' not initialized
  }
  public void check() {
    if(simple == null)
```

```
      System.out.println("not initialized");
    else
      System.out.println("initialized");
  }
  private Simple lazy() {
    if(simple == null) {
      System.out.println("Creating Simple");
      simple = new Simple(s);
    }
    return simple;
  }
  public Simple getSimple() { return lazy(); }
  public String toString() {
    return lazy().toString();
  }
  public void setSimple(String sNew) {
    lazy().setString(sNew);
  }
}

public class E03_Composition {
  public static void main(String args[]) {
    Second second = new Second("Init String");
    second.check();
    System.out.println(second.getSimple());
    second.check();
    System.out.println(second); // toString() call
    second.setSimple("New String");
  }
} ///:~
```

The **Simple** class has some data and methods. The **Second** class performs the lazy initialization through the **lazy( )** method, which is called by all the other methods in order to access the **Simple** object. The **lazy( )** method creates the object if it hasn't already been created, and then returns it.

I've added some print statements to show when the initialization happens, and that it doesn't happen more than once. The program output is:

```
not initialized
```

```
Creating Simple
Init String
initialized
Init String
```

Exercise 4

```
//: c06:E04_NewDetergent.java
/****************** Exercise 4 *****************
 * Inherit a new class from class Detergent.
 * Override scrub() and add a new method called
 * sterilize().
 ***********************************************/
// Depends on TIJ:c06:Detergent.java

public class E04_NewDetergent extends Detergent {
  public void scrub() {
    append("Overriden Scrub");
    super.scrub(); // Doesn't have to be first
  }
  public void sterilize() {
    append("Sterilize");
  }
  public static void main(String args[]) {
    E04_NewDetergent nd = new E04_NewDetergent();
    nd.dilute();
    nd.scrub();
    nd.sterilize();
    System.out.println(nd);
  }
} ///:~
```

So that the program will compile, here is the **Detergent** class from the book:

```
//: c06:Detergent.java
// Inheritance syntax & properties.

class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public String toString() { return s; }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    System.out.println(x);
  }
}

public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call the base-class version.
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    System.out.println(x);
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

Exercise 5

```
//: c06:E05_Cartoon2.java
/****************** Exercise 5 ******************
 * Take the file Cartoon.java and comment out the
 * constructor for the Cartoon class. Explain
 * what happens.
 ***********************************************/
```

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

class Cartoon extends Drawing {
//!  Cartoon() {
//!    System.out.println("Cartoon constructor");
//!  }
}

public class E05_Cartoon2 {
  public static void main(String args[]) {
    new Cartoon();
  }
} ///:~
```

With the **Cartoon** constructor written out explicitly, the output is:

```
Art constructor
Drawing constructor
Cartoon constructor
```

If the **Cartoon** constructor is commented out, the output is:

```
Art constructor
Drawing constructor
```

The compiler synthesizes the default **Cartoon** constructor, and in this generated constructor it automatically calls the base class **Drawing** default constructor, which in turn automatically calls the base class **Art** default constructor.

The compiler will ensure that *some* constructor is called, and if you don't explicitly call a constructor it will call the default constructor for you, if one is available. The default constructor is not available for a class if one or more constructors are explicitly defined, but not the default.

Exercise 6

```
//: c06:E06_Chess2.java
// {CompileByHand} to see that it doesn't compile.
/****************** Exercise 6 *****************
 * Take the file Chess.java and comment out the
 * constructor for the Chess class. Explain what
 * happens.
 ***********************************************/
class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

class Chess extends BoardGame {
  //Chess() {
  //  super(11);
  //  System.out.println("Chess constructor");
  //}
}

public class E06_Chess2 {
  public static void main(String args[]) {
    new Chess();
  }
} ///:~
```

The program won't compile, because the compiler cannot synthesize a default constructor for **Chess**, because **BoardGame** does not have a default constructor that can be used in **Chess**' default constructor. Since **BoardGame** has a defined constructor that takes an argument, the compiler cannot assume that you've simply forgotten to create any constructors and generate one for you.

Because this compilation fails, the **{CompileByHand}** directive takes it out of the build process.

Exercise 7

```
//: c06:E07_AutoDefault.java
/****************** Exercise 7 *****************
 * Prove that default constructors are created
 * for you by the compiler.
 ***********************************************/
public class E07_AutoDefault {
  public static void main(String args[]) {
    new E07_AutoDefault();
  }
} ///:~
```

The mere fact that this compiles is the proof – a constructor is called, and none was defined.

Exercise 8

```
//: c06:E08_ConstructorOrder.java
/****************** Exercise 8 ******************
 * Prove that the base-class constructors are (a)
 * always called, and (b) called before
 * derived-class constructors.
 ***********************************************/
class Base1 {
  public Base1() {
    System.out.println("Base1");
  }
}

class Derived1 extends Base1 {
  public Derived1() {
    System.out.println("Derived1");
  }
}

class Derived2 extends Derived1 {
  public Derived2() {
    System.out.println("Derived2");
  }
}

public class E08_ConstructorOrder {
  public static void main(String args[]) {
    new Derived2();
  }
} ///:~
```

The output is:

```
Base1
Derived1
Derived2
```

Exercise 9

```
//: c06:E09_CallBaseConstructor.java
/****************** Exercise 9 *****************
 * Create a base class with only a nondefault
 * constructor, and a derived class with both a
 * default (no-arg) and nondefault constructor.
 * In the derived-class constructors, call the
 * base-class constructor.
 ***********************************************/
class BaseNonDefault {
  public BaseNonDefault(int i) {}
}

class DerivedTwoConstructors extends BaseNonDefault {
  public DerivedTwoConstructors() {
    super(47);
  }
  public DerivedTwoConstructors(int i) {
    super(i);
  }
}

public class E09_CallBaseConstructor {
  public static void main(String args[]) {
    new DerivedTwoConstructors();
    new DerivedTwoConstructors(74);
  }
} ///:~
```

Exercise 10

```
//: c06:E10_ConstructorOrder.java
/****************** Exercise 10 *****************
 * Create a class called Root that contains an
 * instance of each of the classes (that you also
 * create) named Component1, Component2, and
 * Component3. Derive a class Stem from Root that
 * also contains an instance of each "component."
 * All classes should have default constructors
 * that print a message about that class.
 ***********************************************/
class Component1 {
  public Component1() {
    System.out.println("Component1");
  }
}

class Component2 {
  public Component2() {
    System.out.println("Component2");
  }
}

class Component3 {
  public Component3() {
    System.out.println("Component3");
  }
}
```

```
class Root {
  Component1 c1 = new Component1();
  Component2 c2 = new Component2();
  Component3 c3 = new Component3();
  public Root() {
    System.out.println("Root");
  }
}

class Stem extends Root {
  Component1 c1 = new Component1();
  Component2 c2 = new Component2();
  Component3 c3 = new Component3();
  public Stem() {
    System.out.println("Stem");
  }
}

public class E10_ConstructorOrder {
  public static void main(String args[]) {
    new Stem();
  }
} ///:~
```

The output is:

```
Component1
Component2
Component3
Root
Component1
Component2
Component3
Stem
```

Exercise 11

```
//: c06:E11_ConstructorOrder2.java
/****************** Exercise 11 *****************
 * Modify Exercise 10 so that each class only has
 * nondefault constructors.
 ***********************************************/
class Component1b {
  public Component1b(int i) {
    System.out.println("Component1b");
  }
}

class Component2b {
  public Component2b(int i) {
    System.out.println("Component2b");
  }
}

class Component3b {
  public Component3b(int i) {
    System.out.println("Component3b");
  }
}

class Rootb {
  Component1b c1 = new Component1b(1);
  Component2b c2 = new Component2b(2);
  Component3b c3 = new Component3b(3);
  public Rootb(int i) {
    System.out.println("Rootb");
  }
}

class Stemb extends Rootb {
  Component1b c1 = new Component1b(4);
  Component2b c2 = new Component2b(5);
  Component3b c3 = new Component3b(6);
  public Stemb(int i) {
    super(i);
    System.out.println("Stemb");
  }
}

public class E11_ConstructorOrder2 {
  public static void main(String args[]) {
    new Stemb(47);
  }
} ///:~
```

Exercise 12

```
//: c06:E12_Dispose.java
/****************** Exercise 12 *****************
 * Add a proper hierarchy of dispose() methods to
 * all the classes in Exercise 11.
 ***********************************************/
class Component1c {
  public Component1c(int i) {
    System.out.println("Component1c");
  }
  public void dispose() {
    System.out.println("Component1c dispose");
  }
}

class Component2c {
  public Component2c(int i) {
    System.out.println("Component2c");
  }
  public void dispose() {
    System.out.println("Component2c dispose");
  }
}

class Component3c {
  public Component3c(int i) {
    System.out.println("Component3c");
  }
  public void dispose() {
    System.out.println("Component3c dispose");
  }
}

class Rootc {
  Component1c c1 = new Component1c(1);
  Component2c c2 = new Component2c(2);
  Component3c c3 = new Component3c(3);
  public Rootc(int i) {
    System.out.println("Rootc");
  }
  public void dispose() {
    System.out.println("Rootc dispose");
    c3.dispose();
    c2.dispose();
    c1.dispose();
  }
}

class Stemc extends Rootc {
  Component1c c1 = new Component1c(4);
  Component2c c2 = new Component2c(5);
  Component3c c3 = new Component3c(6);
  public Stemc(int i) {
    super(i);
    System.out.println("Stemc");
  }
  public void dispose() {
    System.out.println("Stemc dispose");
    c3.dispose();
    c2.dispose();
    c1.dispose();
    super.dispose();
  }
}

public class E12_Dispose {
  public static void main(String args[]) {
    new Stemc(47).dispose();
  }
} ///:~
```

It's important to call the **dispose( )** methods in the reverse order of initialization. You can see this in the output:

```
Component1c
Component2c
Component3c
```

```
Rootc
Component1c
Component2c
Component3c
Stemc
Stemc dispose
Component3c dispose
Component2c dispose
Component1c dispose
Rootc dispose
Component3c dispose
Component2c dispose
Component1c dispose
```

Exercise 13

```
//: c06:E13_InheritedOverloading.java
/****************** Exercise 13 *****************
 * Create a class with a method that is
 * overloaded three times. Inherit a new class,
 * add a new overloading of the method, and show
 * that all four methods are available in the
 * derived class.
 ***********************************************/
class ThreeOverloads {
  public void f(int i) {
    System.out.println("f(int i)");
  }
  public void f(char c) {
    System.out.println("f(char c)");
  }
  public void f(double d) {
    System.out.println("f(double d)");
  }
}

class MoreOverloads extends ThreeOverloads {
  public void f(String s) {
    System.out.println("f(String s)");
  }
}

public class E13_InheritedOverloading {
  public static void main(String args[]) {
    MoreOverloads mo = new MoreOverloads();
    mo.f(1);
    mo.f('c');
    mo.f(1.1);
    mo.f("Hello");
  }
} ///:~
```

Exercise 14

```
//: c06:E14_EngineService.java
/****************** Exercise 14 *****************
 * In Car.java add a service() method to Engine
 * and call this method in main().
 ***********************************************/
class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
  public void service() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}
```

```
class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door
    left = new Door(),
    right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
}

public class E14_EngineService {
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
    car.engine.service();
  }
} ///:~
```

Exercise 15

```
//: c06:protect:E15_Protected.java
/****************** Exercise 15 *****************
 * Create a class inside a package. Your class
 * should contain a protected method. Outside of
 * the package, try to call the protected method
 * and explain the results. Now inherit from your
 * class and call the protected method from
 * inside a method of your derived class.
 ***********************************************/
package c06.protect;

public class E15_Protected {
  protected void f() {}
} ///:~
```

```
//: c06:E15_ProtectedTest.java
import c06.protect.*;
```

```
class Derived extends E15_Protected {
  public void g() {
    f(); // Accessible in derived class
  }
}

public class E15_ProtectedTest {
  public static void main(String args[]) {
    //! new E15_Protected().f(); // Cannot access
    new Derived().g();
  }
} ///:~
```

Exercise 16

```
//: c06:E16_Frog.java
/****************** Exercise 16 *****************
 * Create a class called Amphibian. From this,
 * inherit a class called Frog. Put appropriate
 * methods in the base class. In main(), create a
 * Frog and upcast it to Amphibian, and
 * demonstrate that all the methods still work.
 ***********************************************/
class Amphibian {
  public void moveInWater() {
    System.out.println("Moving in Water");
  }
  public void moveOnLand() {
    System.out.println("Moving on Land");
  }
}

class Frog extends Amphibian {}

public class E16_Frog {
  public static void main(String args[]) {
    Amphibian a = new Frog();
    a.moveInWater();
    a.moveOnLand();
  }
} ///:~
```

The output is:

```
Moving in Water
Moving on Land
```

Exercise 17

```
//: c06:E17_Frog2.java
/****************** Exercise 17 *****************
 * Modify Exercise 16 so that Frog overrides the
 * method definitions from the base class
 * (provides new definitions using the same
 * method signatures). Note what happens in
 * main().
 ***********************************************/
class Amphibian2 {
  public void moveInWater() {
    System.out.println("Moving in Water");
  }
  public void moveOnLand() {
    System.out.println("Moving on Land");
  }
}

class Frog2 extends Amphibian2 {
  public void moveInWater() {
    System.out.println("Frog swimming");
  }
  public void moveOnLand() {
    System.out.println("Frog jumping");
  }
}

public class E17_Frog2 {
  public static void main(String args[]) {
    Amphibian2 a = new Frog2();
    a.moveInWater();
    a.moveOnLand();
  }
} ///:~
```

Now the output is

```
Frog swimming
Frog jumping
```

Even though the compiler has a reference to an **Amphibian2**, which might make you think that it would call the **Amphibian2** methods, the **Frog2** methods actually get called. Since **a** is indeed a reference to a **Frog2**, this is the behavior you want. That's polymorphism – the right behavior happens even if you are talking to a base-class reference.

Exercise 18

```
//: c06:E18_FinalFields.java
/****************** Exercise 18 *****************
 * Create a class with a static final field and a
 * final field and demonstrate the difference
 * between the two.
 ***********************************************/
class SelfCounter {
  private static int count = 0;
  private int id = count++;
  public String toString() {
    return "SelfCounter " + id;
  }
}

class WithFinalFields {
  final SelfCounter scf = new SelfCounter();
  static final SelfCounter scsf = new SelfCounter();
  public String toString() {
    return "scf = " + scf + "\nscsf = " + scsf;
  }
}

public class E18_FinalFields {
  public static void main(String args[]) {
    System.out.println("First object:");
    System.out.println(new WithFinalFields());
    System.out.println("Second object:");
    System.out.println(new WithFinalFields());
  }
} ///:~
```

The output is:

```
First object:
scf = SelfCounter 1
scsf = SelfCounter 0
Second object:
scf = SelfCounter 2
scsf = SelfCounter 0
```

Note that the **static final** field was initialized first, because it got the value of zero as the **WithFinalFields** class was being loaded. And in both instances of **WithFinalFields**, the **static final** field has the same value because it is only initialized upon class loading, whereas the regular **final** field gets a different value for each instance.

Exercise 19

```
//: c06:E19_BlankFinal.java
/****************** Exercise 19 *****************
 * Create a class with a blank final reference to
 * an object. Perform the initialization of the
 * blank final inside all constructors.
 * Demonstrate the guarantee that the final must
 * be initialized before use, and that it cannot
 * be changed once initialized.
 ***********************************************/
class WithBlankFinal  {
  private final Integer i;
  // Without this constructor, you'll get a compiler error:
  // "variable i might not have been initialized"
  public WithBlankFinal(int ii) {
    i = new Integer(ii);
  }
  public Integer geti() {
    // This won't compile. The error is:
```

```
    // "cannot assign a value to final variable i"
    // if(i == null)
    //   i = new Integer(47);
    return i;
  }
}

public class E19_BlankFinal {
  public static void main(String args[]) {
    WithBlankFinal wbf = new WithBlankFinal(10);
    System.out.println(wbf.geti());
  }
} ///:~
```

Exercise 20

```
//: c06:E20_FinalMethod.java
/****************** Exercise 20 ****************
 * Create a class with a final method. Inherit
 * from that class and attempt to override that
 * method.
 ***********************************************/
class WithFinal {
  final void f() {}
}

public class E20_FinalMethod extends WithFinal {
  // Compiler error -- cannot override final method:
  //! void f() {}
  public static void main(String args[]) {
    new E20_FinalMethod();
  }
} ///:~
```

Exercise 21

```
//: c06:E21_FinalClass.java
/****************** Exercise 21 ****************
 * Create a final class and attempt to inherit
 * from it.
 ***********************************************/
final class FinalClass {}

public class E21_FinalClass
  // Compiler error -- cannot inherit from
  // final class:
  //! extends FinalClass
 {
  public static void main(String args[]) {
  }
} ///:~
```

Exercise 22

```
//: c06:E22_ClassLoading.java
/***************** Exercise 22 ***************
 * Prove that class loading takes place only
 * once. Prove that loading may be caused by
 * either the creation of the first instance of
 * that class, or the access of a static member.
 ***********************************************/
class LoadTest {
  // The static clause is executed
  // upon class loading:
  static {
    System.out.println("Loading LoadTest");
  }
  static void staticMember() {}
}

public class E22_ClassLoading {
  public static void main(String args[]) {
    System.out.println("Calling static member");
    LoadTest.staticMember();
    System.out.println("Creating an object");
    new LoadTest();
  }
} ///:~
```

When you run this program, you get the output:

```
Calling static member
Loading LoadTest
Creating an object
```

Now try modifying the code to put the object creation before the **static** member call, and you'll see that the object creation will cause the object to be loaded. Actually, a constructor is a **static** method even though you don't explicitly use the **static** keyword when defining it.

Exercise 23

```
//: c06:E23_JapaneseBeetle.java
/****************** Exercise 23 ****************
 * In Beetle.java, inherit a specific type of
 * beetle from class Beetle, following the same
 * format as the existing classes. Trace and
 * explain the output.
 ***********************************************/
class Insect {
  int i = 9;
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 = prt("static Insect.x1 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

class Beetle extends Insect {
  int k = prt("Beetle.k initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 = prt("static Beetle.x2 initialized");
}


class JapaneseBeetle extends Beetle {
  int m = prt("JapaneseBeetle.m initialized");
  JapaneseBeetle() {
    System.out.println("m = " + m);
    System.out.println("j = " + j);
  }
  static int x3 = prt(
    "static JapaneseBeetle.x3 initialized");
}

public class E23_JapaneseBeetle {
  public static void main(String args[]) {
    new JapaneseBeetle();
  }
} ///:~
```

The output is:

```
static Insect.x1 initialized
static Beetle.x2 initialized
static JapaneseBeetle.x3 initialized
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
JapaneseBeetle.m initialized
m = 47
j = 39
```

The **static** variables are initialized when the class is loaded; notice that the base class of the entire hierarchy is loaded first, then the next-derived class, finally the most-derived class. Then the object is created, and the non-**static** members are initialized, also starting at the root class.

Chapter 7

Exercise 1

```
//: c07:E01_NewShapeMethod.java
/****************** Exercise 1 *****************
 * Add a new method in the base class of
 * Shapes.java that prints a message, but don't
 * override it in the derived classes. Explain
 * what happens. Now override it in one of the
 * derived classes but not the others, and see
 * what happens. Finally, override it in all the
 * derived classes.
 ***********************************************/
class Shape {
  void draw() {}
  void erase() {}
  void print() {
    System.out.println("Base class print()");
  }
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()");
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
  void print() {
    System.out.println("Circle.print()");
  }
}

class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
  void print() {
    System.out.println("Square.print()");
  }
}

class Triangle extends Shape {
  void draw() {
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }
  void print() {
    System.out.println("Triangle.print()");
  }
}

public class E01_NewShapeMethod {
  public static void main(String args[]) {
    Shape[] s = {
      new Circle(), new Square(), new Triangle(),
    };
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++) {
      s[i].draw();
      s[i].erase();
      s[i].print();
    }
  }
} ///:~
```

The above code is the final version, with **print( )** overriden in all classes.

With **print( )** defined only in the base class, the output is:

```
Circle.draw()
Circle.erase()
Base class print()
```

```
Square.draw()
Square.erase()
Base class print()
Triangle.draw()
Triangle.erase()
Base class print()
```

Since the base-class version is not being overridden, it is used everywhere.

With **print( )** overridden in **Circle**, the output is:

```
Circle.draw()
Circle.erase()
Circle.print()
Square.draw()
Square.erase()
Base class print()
Triangle.draw()
Triangle.erase()
Base class print()
```

The overridden version is used in **Circle**, and the default base-class version is used everywhere else.

If **print( )** is overridden everywhere, the result is:

```
Circle.draw()
Circle.erase()
Circle.print()
Square.draw()
Square.erase()
Square.print()
Triangle.draw()
Triangle.erase()
Triangle.print()
```

Thus, the overridden version will always be used if it is available.

Exercise 2

```
//: c07:E02_NewShapeType.java
/****************** Exercise 2 *****************
```

```
 * Add a new type of Shape to Shapes.java and
 * verify in main() that polymorphism works for
 * your new type as it does in the old types.
 ***********************************************/
class Tetrahedron extends Shape {
  void draw() {
    System.out.println("Tetrahedron.draw()");
  }
  void erase() {
    System.out.println("Tetrahedron.erase()");
  }
  void print() {
    System.out.println("Tetrahedron.print()");
  }
}

public class E02_NewShapeType {
  public static void main(String args[]) {
    Shape[] s = {
      new Circle(), new Square(), new Triangle(),
      new Tetrahedron()
    };
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++) {
      s[i].draw();
      s[i].erase();
      s[i].print();
    }
  }
} ///:~
```

Since the other shape definitions are in the same directory (default package) we can just add the new shape and override the methods. The code in the **for** loop is unchanged from the previous example. The output is:

```
Circle.draw()
Circle.erase()
Circle.print()
Square.draw()
Square.erase()
Square.print()
Triangle.draw()
Triangle.erase()
Triangle.print()
Tetrahedron.draw()
Tetrahedron.erase()
Tetrahedron.print()
```

Exercise 3

```
//: c07:E03_MusicToString.java
/****************** Exercise 3 *****************
 * Change Music3.java so that what() becomes the
 * root Object method toString(). Try printing
 * the Instrument objects using
 * System.out.println() (without any casting).
 ***********************************************/

class E03_Instrument {
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String toString() {
    return "Instrument";
  }
  public void adjust() {}
}

class E03_Wind extends E03_Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String toString() {
    return "Wind";
  }
  public void adjust() {}
}

class E03_Percussion extends E03_Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String toString() {
    return "Percussion";
  }
  public void adjust() {}
}

class E03_Stringed extends E03_Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String toString() {
    return "Stringed";
  }
  public void adjust() {}
}

class E03_Brass extends E03_Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class E03_Woodwind extends E03_Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String toString() {
    return "Woodwind";
  }
}

public class E03_MusicToString {
  static E03_Instrument[] orchestra = {
    new E03_Wind(),
    new E03_Percussion(),
    new E03_Stringed(),
    new E03_Brass(),
    new E03_Woodwind()
  };
  public static void printAll(E03_Instrument[] orch) {
    for(int i = 0; i < orch.length; i++)
      System.out.println(orch[i]);
  }
  public static void main(String args[]) {
    printAll(orchestra);
  }
} ///:~
```

The output is:

```
Wind
Percussion
Stringed
Wind
Woodwind
```

Exercise 4

```
//: c07:E04_NewInstrument.java
/****************** Exercise 4 *****************
 * Add a new type of Instrument to Music3.java
 * and verify that polymorphism works for your
 * new type.
 ***********************************************/
class Electronic extends Instrument {
  public void play(Note n) {
    System.out.println("Electronic.play() " + n);
  }
  public String what() { return "Electronic"; }
  public void adjust() {}
}

public class E04_NewInstrument {
  static Instrument[] orchestra = {
    new Wind(),
    new Percussion(),
    new Stringed(),
    new Brass(),
    new Woodwind(),
    new Electronic()
  };
  public static void main(String args[]) {
    for(int i = 0; i < orchestra.length; i++) {
      orchestra[i].play(Note.MIDDLE_C);
      orchestra[i].adjust();
      System.out.println(orchestra[i].what());
    }
  }
} ///:~
```

The output is:

```
Wind.play() Middle C
Wind
Percussion.play() Middle C
Percussion
Stringed.play() Middle C
Stringed
Brass.play() Middle C
Brass.adjust()
Wind
Woodwind.play() Middle C
Woodwind
Electronic.play() Middle C
Electronic
```

Exercise 5

```
//: c07:E05_RandomInstruments.java
/****************** Exercise 5 *****************
 * Modify Music3.java so that it randomly creates
 * Instrument objects the way Shapes.java does.
 ***********************************************/

class InstrumentGenerator {
  java.util.Random gen = new java.util.Random();
  public Instrument next() {
    switch(gen.nextInt(6)) {
      default:
      case 0: return new Wind();
      case 1: return new Percussion();
      case 2: return new Stringed();
      case 3: return new Brass();
      case 4: return new Woodwind();
      case 5: return new Electronic();
    }
  }
}

public class E05_RandomInstruments {
  public static void main(String args[]) {
    InstrumentGenerator gen = new InstrumentGenerator();
    for(int i = 0; i < 20; i++)
      System.out.println(gen.next());
  }
} ///:~
```

The idea of a *generator* is a common one. Every time you call it, it produces a new value, but when you call it you don't give it any parameters.

The random number generator method **nextInt( )** produces a random value between zero (inclusive) and the value of the argument (exclusive), so it will always be within the range of the array.

One of the flaws with this design is managing the **case** statement, which is a bit awkward and error-prone. It would be much nicer if we could just index into an array of objects that could themselves generate the different kinds of instruments. This is possible if we use some features of the **Class** object. Here's the more elegant solution (that I *didn't* expect you to get!):

```
//: c07:E05_RandomInstruments2.java
// A more sophisticated solution using features
// you'll learn about in later chapters.

class InstrumentGenerator2 {
  java.util.Random gen = new java.util.Random();
  Class instruments[] = {
    Wind.class,
    Percussion.class,
    Stringed.class,
    Brass.class,
    Woodwind.class,
    Electronic.class,
  };
  public Instrument next() {
    try {
      int idx = Math.abs(gen.nextInt(instruments.length));
      return (Instrument) instruments[idx].newInstance();
    } catch(Exception e) {
      System.out.println("e = " + e);
      throw new RuntimeException(
        "Cannot Create Instrument");
    }
  }
}

public class E05_RandomInstruments2 {
  public static void main(String args[]) {
    InstrumentGenerator2 gen = new InstrumentGenerator2();
    for(int i = 0; i < 20; i++)
      System.out.println(gen.next());
  }
} ///:~
```

References to **Class** objects for each type of instrument can be produced using the **.class** that you see in the array. **Class** objects have a method **newInstance( )** which will create objects of their particular class, but these will throw exceptions. Here, I create and throw a **RuntimeException** (you'll learn about exceptions in chapter 9) because I consider it a programming error, and thus doesn't need to be guarded against in your code (that is, your code shouldn't have to catch programming errors).

Note the benefit of this design – if you need to add a new type to the system, you only need to add it in the **Class** array; the rest of the code takes care of itself.

Exercise 6

```
//: c07:E06_Rodents.java
/****************** Exercise 6 *****************
 * Create an inheritance hierarchy of Rodent:
 * Mouse, Gerbil, Hamster, etc. In the base
 * class, provide methods that are common to all
 * Rodents, and override these in the derived
 * classes to perform different behaviors
 * depending on the specific type of Rodent.
 * Create an array of Rodent, fill it with
 * different specific types of Rodents, and call
 * your base-class methods to see what happens.
 ***********************************************/

class Rodent {
  public void hop() {
    System.out.println("Rodent hopping");
  }
  public void scurry() {
    System.out.println("Rodent scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Rodents");
  }
  public String toString() {
    return "Rodent";
  }
}

class Mouse extends Rodent {
  public void hop() {
    System.out.println("Mouse hopping");
  }
  public void scurry() {
    System.out.println("Mouse scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Mice");
  }
  public String toString() {
    return "Mouse";
  }
}

class Gerbil extends Rodent {
  public void hop() {
    System.out.println("Gerbil hopping");
  }
  public void scurry() {
    System.out.println("Gerbil scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Gerbils");
  }
  public String toString() {
    return "Gerbil";
  }
}

class Hamster extends Rodent {
  public void hop() {
    System.out.println("Hamster hopping");
  }
  public void scurry() {
    System.out.println("Hamster scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Hamsters");
  }
  public String toString() {
    return "Hamster";
  }
}

public class E06_Rodents {
  public static void main(String args[]) {
    Rodent[] rodents = {
      new Mouse(),
      new Gerbil(),
      new Hamster(),
    };
    for(int i = 0; i < rodents.length; i++) {
      rodents[i].hop();
      rodents[i].scurry();
      rodents[i].reproduce();
      System.out.println(rodents[i]);
    }
  }
} ///:~
```

The output is:

```
Mouse hopping
Mouse scurrying
Making more Mice
Mouse
Gerbil hopping
Gerbil scurrying
Making more Gerbils
Gerbil
Hamster hopping
Hamster scurrying
Making more Hamsters
Hamster
```

Exercise 7

```
//: c07:E07_AbstractRodent.java
/****************** Exercise 7 *****************
 * Modify Exercise 6 so that Rodent is an
 * abstract class. Make the methods of Rodent
 * abstract whenever possible.
 ***********************************************/
abstract class Rodent2 {
  public abstract void hop();
  public abstract void scurry();
  public abstract void reproduce();
}

class Mouse2 extends Rodent2 {
  public void hop() {
    System.out.println("Mouse hopping");
  }
  public void scurry() {
    System.out.println("Mouse scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Mice");
  }
  public String toString() {
    return "Mouse";
  }
}

class Gerbil2 extends Rodent2 {
  public void hop() {
    System.out.println("Gerbil hopping");
  }
  public void scurry() {
    System.out.println("Gerbil scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Gerbils");
  }
  public String toString() {
    return "Gerbil";
  }
}

class Hamster2 extends Rodent2 {
  public void hop() {
    System.out.println("Hamster hopping");
  }
  public void scurry() {
    System.out.println("Hamster scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Hamsters");
  }
  public String toString() {
    return "Hamster";
  }
```

```
}

public class E07_AbstractRodent {
  public static void main(String args[]) {
    Rodent2[] rodents = {
      new Mouse2(),
      new Gerbil2(),
      new Hamster2(),
    };
    for(int i = 0; i < rodents.length; i++) {
      rodents[i].hop();
      rodents[i].scurry();
      rodents[i].reproduce();
      System.out.println(rodents[i]);
    }
  }
} ///:~
```

This produces the same output as the previous example. Note that **toString( )** is a method of the root class **Object** and thus can be left out of the **abstract** base class.

Exercise 8

```
//: c07:E08_Abstract.java
/****************** Exercise 8 *****************
 * Create a class as abstract without including
 * any abstract methods, and verify that you
 * cannot create any instances of that class.
 ***********************************************/

abstract class NoAbstractMethods {
  void f() { System.out.println("f()"); }
}

public class E08_Abstract {
  public static void main(String args[]) {
    // Compile-time error: class is abstract,
    // cannot be instantiated.
    //! new NoAbstractMethods();
  }
} ///:~
```

Exercise 9

```
//: c07:E09_Pickle.java
/****************** Exercise 9 *****************
 * Add class Pickle to Sandwich.java.
 ***********************************************/
class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Pickle {
  Pickle() { System.out.println("Pickle()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Pickle p = new Pickle();
  Sandwich() {
    System.out.println("Sandwich()");
  }
}

public class E09_Pickle {
  public static void main(String args[]) {
    new Sandwich();
  }
} ///:~
```

The output is:

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Pickle()
Sandwich()
```

Note that the **Pickle** object is created in order, after the other member objects.

Exercise 10

```
//: c07:E10_RodentInitialization.java
/****************** Exercise 10 *****************
 * Modify Exercise 6 so that it demonstrates the
 * order of initialization of the base classes
 * and derived classes. Now add member objects to
 * both the base and derived classes, and show
 * the order in which their initialization occurs
 * during construction.
 ***********************************************/
class Member {
  public Member(String id) {
    System.out.println("Member constructor " + id);
  }
}

class Rodent3 {
  Member m1 = new Member("r1"), m2 = new Member("r2");
  public Rodent3() {
    System.out.println("Rodent constructor");
  }
  public void hop() {
    System.out.println("Rodent hopping");
  }
  public void scurry() {
    System.out.println("Rodent scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Rodents");
  }
  public String toString() {
    return "Rodent";
  }
}

class Mouse3 extends Rodent3 {
  Member m1 = new Member("m1"), m2 = new Member("m2");
  public Mouse3() {
    System.out.println("Mouse constructor");
  }
  public void hop() {
    System.out.println("Mouse hopping");
  }
  public void scurry() {
    System.out.println("Mouse scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Mice");
  }
  public String toString() {
    return "Mouse";
  }
```

```
}

class Gerbil3 extends Rodent3 {
  Member m1 = new Member("g1"), m2 = new Member("g2");
  public Gerbil3() {
    System.out.println("Gerbil constructor");
  }
  public void hop() {
    System.out.println("Gerbil hopping");
  }
  public void scurry() {
    System.out.println("Gerbil scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Gerbils");
  }
  public String toString() {
    return "Gerbil";
  }
}

class Hamster3 extends Rodent3 {
  Member m1 = new Member("h1"), m2 = new Member("h2");
  public Hamster3() {
    System.out.println("Hamster constructor");
  }
  public void hop() {
    System.out.println("Hamster hopping");
  }
  public void scurry() {
    System.out.println("Hamster scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Hamsters");
  }
  public String toString() {
    return "Hamster";
  }
}

public class E10_RodentInitialization {
  public static void main(String args[]) {
    new Hamster3();
  }
} ///:~
```

The output for creating the **Hamster3** object is:

```
Member constructor r1
Member constructor r2
Rodent constructor
Member constructor h1
Member constructor h2
Hamster constructor
```

The base class is initialized first, starting with the member objects in their order of definition, then the derived class starting with its member objects.

Exercise 11

```
//: c07:E11_MethodCalls.java
/****************** Exercise 11 *****************
 * Create a base class with two methods. In the
 * first method, call the second method. Inherit
 * a class and override the second method. Create
 * an object of the derived class, upcast it to
 * the base type, and call the first method.
 * Explain what happens.
 ***********************************************/
class TwoMethods {
  public void m1() {
    System.out.println("Inside m1, calling m2");
    m2();
  }
  public void m2() {
    System.out.println("Inside m2");
  }
}

class Inherited extends TwoMethods {
  public void m2() {
    System.out.println("Inside Inherited.m2");
  }
}

public class E11_MethodCalls {
  public static void main(String args[]) {
    TwoMethods x = new Inherited();
    x.m1();
  }
} ///:~
```

The output is:

```
Inside m1, calling m2
Inside Inherited.m2
```

The first method isn't overriden, but it calls the second method, which is. Java will always use the most-derived method it can find for the object type. If you're not thinking about this, it can be surprising. However, it can also be very powerful – this is the way the *Template Method* design pattern works (see *Thinking in Patterns*, downloadable from www.BruceEckel.com).

Exercise 12

```
//: c07:E12_Initialization.java
/****************** Exercise 12 *****************
 * Create a base class with an abstract print()
 * method that is overridden in a derived class.
 * The overridden version of the method prints
 * the value of an int variable defined in the
 * derived class. At the point of definition of
 * this variable, give it a nonzero value. In the
 * base-class constructor, call this method. In
 * main(), create an object of the derived type,
 * and then call its print() method. Explain the
 * results.
 ***********************************************/
class BaseWithPrint {
  public BaseWithPrint() { print(); }
  public void print() {
    System.out.println("BaseWithPrint.print");
  }
}

class DerivedWithPrint extends BaseWithPrint {
  int i = 47;
  public void print() {
    System.out.println("i = " + i);
  }
}

public class E12_Initialization {
  public static void main(String args[]) {
    DerivedWithPrint dp = new DerivedWithPrint();
    dp.print();
  }
} ///:~
```

The output is:

```
i = 0
i = 47
```

When the base-class constructor is called, the derived-class initialization has not yet been run, so the value of **i** is the default which comes from the java virtual machine automatically wiping the bits of the object to zero right after the storage has been allocated.

This exercise is intended to point out the dangers of calling methods inside of constructors. Although it is certainly justified at times, you should be careful because these methods may depend on some derived initialization that hasn't yet taken place. The safest approach is to do as little as possible to set the object into a known good state, and then do any other operations separately from the constructor.

Exercise 13

```
//: c07:E13_Starship.java
/****************** Exercise 13 *****************
 * Following the example in Transmogrify.java,
 * create a Starship class containing an
 * AlertStatus reference that can indicate three
 * different states. Include methods to change
 * the states.
 **********************************************/
class AlertStatus {
  public String getStatus() { return "None"; }
}

class RedAlertStatus extends AlertStatus {
  public String getStatus() { return "Red"; };
}

class YellowAlertStatus extends AlertStatus {
  public String getStatus() { return "Yellow"; };
}

class GreenAlertStatus extends AlertStatus {
  public String getStatus() { return "Green"; };
}

class Starship {
  private AlertStatus status = new GreenAlertStatus();
  public void setStatus(AlertStatus istatus) {
    status = istatus;
  }
  public String toString() { return status.getStatus(); }
}

public class E13_Starship {
  public static void main(String args[]) {
    Starship eprise = new Starship();
    System.out.println(eprise);
    eprise.setStatus(new YellowAlertStatus());
    System.out.println(eprise);
    eprise.setStatus(new RedAlertStatus());
    System.out.println(eprise);
  }
} ///:~
```

There are three different variations of the **AlertStatus** class, representing the different behaviors that a **Starship** can be in, depending on its state. The **Starship** class holds a reference to an **AlertStatus** object, and the **setStatus( )** method allows you to change this reference, effectively changing the behavior of the **Starship**.

The output is:

```
Green
Yellow
Red
```

The reason this is an example of the *State* design pattern is that the object behaves differently depending on what state it is in.

Exercise 14

```java
//: c07:E14_AbstractBase.java
/****************** Exercise 14 *****************
 * Create an abstract class with no methods.
 * Derive a class and add a method. Create a
 * static method that takes a reference to the
 * base class, downcasts it to the derived class,
 * and calls the method. In main(), demonstrate
 * that it works. Now put the abstract
 * declaration for the method in the base class,
 * thus eliminating the need for the downcast.
 ***********************************************/
abstract class NoMethods {}

class Extended1 extends NoMethods {
  public void f() {
    System.out.println("Extended1.f");
  }
}

abstract class WithMethods {
  abstract public void f();
}

class Extended2 extends WithMethods {
  public void f() {
    System.out.println("Extended2.f");
  }
}

public class E14_AbstractBase {
  public static void test1(NoMethods nm) {
    // Must downcast to access f():
    ((Extended1)nm).f();
  }
  public static void test2(WithMethods wm) {
    // No downcast necessary:
    wm.f();
  }
  public static void main(String args[]) {
    NoMethods nm = new Extended1();
    test1(nm);
    WithMethods wm = new Extended2();
    test2(wm);
  }
} ///:~
```

Both versions are shown here, and in **test1( )** you can see how a downcast is necessary in order to call **f( )** (take out the downcast if you don't believe it). In **test2( )**, no downcast is necessary because the method **f( )** is defined in the base class.

The output is:

```
Extended1.f
Extended2.f
```

Chapter 8

Exercise 1

```
//: c08:E01_ImplicitStaticFinal.java
/****************** Exercise 1 ******************
 * Prove that the fields in an interface are
 * implicitly static and final.
 ***********************************************/
interface StaticFinalTest {
  String RED = "Red";
}

class Test implements StaticFinalTest {
  public Test() {
    // Compile-time error: cannot assign a value
    // to final variable RED:
    //!  RED = "Blue";
  }
}

public class E01_ImplicitStaticFinal {
  public static void main(String args[]) {
    // Accessing as a static field:
    System.out.println("StaticFinalTest.RED = "
                       + StaticFinalTest.RED);
  }
} ///:~
```

The compiler tells you that **RED** is a final variable when you try to assign a value to it, and it is clearly **static** because you can access it using the appropriate **static** syntax.

Exercise 2

```
//: c08:E02_ImplementInterface.java
/****************** Exercise 2 ******************
```

```
 * Create an interface containing three methods,
 * in its own package. Implement the interface in
 * a different package.
 ***********************************************/
import c08.ownpackage.*;

class ImplementInterface implements AnInterface {
  public void f() {
    System.out.println("ImplementInterface.f");
  }
  public void g() {
    System.out.println("ImplementInterface.g");
  }
  public void h() {
    System.out.println("ImplementInterface.h");
  }
}

public class E02_ImplementInterface {
  public static void main(String args[]) {
    ImplementInterface imp =
      new ImplementInterface();
    imp.f();
    imp.g();
    imp.h();
  }
} ///:~
```

Note that even though the elements of an interface are automatically public, the interface itself is not, and must be explicitly declared public in order to use it outside of its package:

```
//: c08:ownpackage:AnInterface.java
package c08.ownpackage;

public interface AnInterface {
  void f();
  void g();
  void h();
} ///:~
```

Exercise 3

```
//: c08:E03_InterfacePublicMethods.java
/****************** Exercise 3 *****************
 * Prove that all the methods in an interface are
 * automatically public.
 ***********************************************/
import c08.ownpackage.*;

public class E03_InterfacePublicMethods
  implements AnInterface {
  // Compile-time error messages for each one
  // of these, stating that you cannot reduce
  // the visibility from the base class' public
  // method to a less public access. In this case
  // default or package access.
//!  void f() {}
//!  void g() {}
//!  void h() {}
  // Compiles OK:
  public void f() {}
  public void g() {}
  public void h() {}
  public static void main(String args[]) {
    new E03_InterfacePublicMethods();
  }
} ///:~
```

Exercise 4

```
//: c08:E04_FastFood.java
/****************** Exercise 4 *****************
 * In c07:Sandwich.java, create an interface
 * called FastFood (with appropriate methods) and
 * change Sandwich so that it also implements
 * FastFood.
 ***********************************************/

interface FastFood {
  void rushOrder();
  void gobble();
}

class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

class Sandwich extends PortableLunch
 implements FastFood {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public void rushOrder() {
    System.out.println("Rushing your sandwich order");
  }
  public void gobble() {
    System.out.println("Chomp! Snort! Gobble!");
  }
}

public class E04_FastFood {
  public static void main(String args[]) {
    Sandwich burger = new Sandwich();
    System.out.println("Fries with that?");
    System.out.println("Super Size?");
    burger.rushOrder();
    burger.gobble();
  }
} ///:~
```

The output is:

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
Fries with that?
Super Size?
Rushing your sandwich order
Chomp! Snort! Gobble!
```

Exercise 5

```
//: c08:E05_InterfaceInheritance.java
/****************** Exercise 5 ******************
 * Create three interfaces, each with two
 * methods. Inherit a new interface from the
 * three, adding a new method. Create a class by
 * implementing the new interface and also
 * inheriting from a concrete class. Now write
 * four methods, each of which takes one of the
 * four interfaces as an argument. In main(),
 * create an object of your class and pass it to
 * each of the methods.
 **********************************************/
interface Interface1 {
  void f1();
  void g1();
}

interface Interface2 {
  void f2();
  void g2();
}

interface Interface3 {
  void f3();
  void g3();
}

interface Multiple
  extends Interface1, Interface2, Interface3 {
  void h();
}

class Concrete {
  String s;
  public Concrete(String s) { this.s = s; }
}

class All extends Concrete implements Multiple {
  public All() { super("All"); }
  public void h() { System.out.println("All.h"); }
  public void f1() { System.out.println("All.f1"); }
  public void g1() { System.out.println("All.g1"); }
  public void f2() { System.out.println("All.f2"); }
  public void g2() { System.out.println("All.g2"); }
  public void f3() { System.out.println("All.f3"); }
  public void g3() { System.out.println("All.g3"); }
}

public class E05_InterfaceInheritance {
  public void takes1(Interface1 i) {
    i.f1();
    i.g1();
  }
  public void takes2(Interface2 i) {
    i.f2();
    i.g2();
  }
  public void takes3(Interface3 i) {
    i.f3();
    i.g3();
  }
  public void takesAll(All a) {
    a.f1();
    a.g1();
    a.f2();
    a.g2();
    a.f3();
    a.g3();
    a.h();
  }
  public static void main(String args[]) {
    E05_InterfaceInheritance ii =
      new E05_InterfaceInheritance();
    All a = new All();
    ii.takes1(a);
    ii.takes2(a);
    ii.takes3(a);
    ii.takesAll(a);
  }
} ///:~
```

Exercise 6

```
//: c08:E06_AbstractsAndInterfaces.java
/****************** Exercise 6 ******************
 * Modify Exercise 5 by creating an abstract
 * class and inheriting that into the derived
 * class.
 ***********************************************/
abstract class Abstract {
  String s;
  public Abstract(String s) { this.s = s; }
  abstract void af();
}

class All2 extends Abstract implements Multiple {
  public All2() { super("All2"); }
  public void af() { System.out.println("All.af"); }
  public void f1() { System.out.println("All.f1"); }
  public void g1() { System.out.println("All.g1"); }
  public void f2() { System.out.println("All.f2"); }
  public void g2() { System.out.println("All.g2"); }
  public void f3() { System.out.println("All.f3"); }
  public void g3() { System.out.println("All.g3"); }
  public void h() { System.out.println("All2.h"); }
}

public class E06_AbstractsAndInterfaces {
  public void takes1(Interface1 i) {
    i.f1();
    i.g1();
  }
  public void takes2(Interface2 i) {
    i.f2();
    i.g2();
  }
  public void takes3(Interface3 i) {
    i.f3();
    i.g3();
  }
  public void takesAll(All2 a) {
    a.f1();
    a.g1();
    a.f2();
    a.g2();
    a.f3();
    a.g3();
    a.h();
  }
  public void takesAbstract(Abstract a) {
    a.af();
  }
  public static void main(String args[]) {
    E06_AbstractsAndInterfaces ii =
      new E06_AbstractsAndInterfaces();
    All2 a = new All2();
    ii.takes1(a);
    ii.takes2(a);
    ii.takes3(a);
    ii.takesAll(a);
    ii.takesAbstract(a);
  }
} ///:~
```

Exercise 7

```
//: c08:E07_Playable.java
/****************** Exercise 7 ******************
 * Modify Music5.java by adding a Playable interface. Move
 * the play() declaration from Instrument to Playable. Add
 * Playable to the derived classes by including it in the
 * implements list. Change tune() so that it takes a
 * Playable instead of an Instrument.
 ***********************************************/
interface Instrument {
  int i = 5;
  String what();
  void adjust();
}

interface Playable {
  void play();
}

class Wind implements Instrument, Playable {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}
```

```
class Percussion implements Instrument, Playable {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument, Playable {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

class Music5 {
  static void tune(Playable i) {
    // ...
    i.play();
  }
  static void tuneAll(Playable[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
}

public class E07_Playable {
  public static void main(String args[]) {
    // Upcasting during addition to the array:
    Playable[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind(),
    };
    Music5.tuneAll(orchestra);
  }
} ///:~
```

Exercise 8

```
//: c08:E08_RodentInterface.java
/****************** Exercise 8 ******************
 * Change Exercise 6 in Chapter 7 so that Rodent
 * is an interface.
 ***********************************************/
interface Rodent {
  public void hop();
  public void scurry();
  public void reproduce();
}

class Mouse implements Rodent {
  public void hop() {
    System.out.println("Mouse hopping");
  }
  public void scurry() {
    System.out.println("Mouse scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Mice");
  }
  public String toString() { return "Mouse"; }
```

```java
}

class Gerbil implements Rodent {
  public void hop() {
    System.out.println("Gerbil hopping");
  }
  public void scurry() {
    System.out.println("Gerbil scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Gerbils");
  }
  public String toString() { return "Gerbil"; }
}

class Hamster implements Rodent {
  public void hop() {
    System.out.println("Hamster hopping");
  }
  public void scurry() {
    System.out.println("Hamster scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Hamsters");
  }
  public String toString() { return "Hamster"; }
}

public class E08_RodentInterface {
  public static void main(String args[]) {
    Rodent[] rodents = {
      new Mouse(),
      new Gerbil(),
      new Hamster(),
    };
    for(int i = 0; i < rodents.length; i++) {
      rodents[i].hop();
      rodents[i].scurry();
      rodents[i].reproduce();
      System.out.println(rodents[i]);
    }
  }
} ///:~
```

Exercise 9

```
//: c08:E09_CanClimb.java
/****************** Exercise 9 ******************
 * In Adventure.java, add an interface called
 * CanClimb, following the form of the other
 * interfaces.
 ***********************************************/
interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

interface CanClimb {
  void climb();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
  implements CanFight, CanSwim, CanFly, CanClimb {
  public void swim() {}
  public void fly() {}
  public void climb() {}
}

public class E09_CanClimb {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void z(CanClimb x) { x.climb(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    z(h); // Treat it as a CanClimb
    w(h); // Treat it as an ActionCharacter
  }
} ///:~
```

Exercise 10

```
//: c08:E10_UsingMonth.java
/****************** Exercise 10 *****************
 * Write a program that imports and uses
 * Month.java.
 ***********************************************/
import c08.month.*;

public class E10_UsingMonth {
  public static void main(String args[]) {
    for(int i = 0; i < Month.month.length; i++) {
      System.out.println(
        "Month.month[i] = " + Month.month[i]);
    }
  }
} ///:~
```

Below is Month.java from Thinking in Java, 3rd edition

```
//: c08:month:Month.java
// A more robust enumeration system.
package c08.month;

public final class Month {
  private String name;
  private Month(String nm) { name = nm; }
  public String toString() { return name; }
  public static final Month
    JAN = new Month("January"),
    FEB = new Month("February"),
    MAR = new Month("March"),
    APR = new Month("April"),
    MAY = new Month("May"),
    JUN = new Month("June"),
    JUL = new Month("July"),
    AUG = new Month("August"),
    SEP = new Month("September"),
    OCT = new Month("October"),
    NOV = new Month("November"),
    DEC = new Month("December");
  public static final Month[] month =  {
    JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public static final Month number(int ord) {
    return month[ord - 1];
  }
} ///:~
```

Exercise 11

```
//: c08:E11_DaysOfWeek.java
/****************** Exercise 11 *****************
 * Following the example given in Month.java,
 * create an enumeration of days of the week.
 ***********************************************/
final class Day {
  private String name;
  private int order;
  private Day(int ord, String nm) {
    order = ord;
    name = nm;
  }
  public String toString() { return name; }
  public final static Day
    MONDAY = new Day(1, "Monday"),
    TUESDAY = new Day(2, "Tuesday"),
    WEDNESDAY = new Day(3, "Wednesday"),
    THURSDAY = new Day(4, "Thursday"),
    FRIDAY = new Day(5, "Friday"),
    SATURDAY = new Day(6, "Saturday"),
    SUNDAY = new Day(7, "Sunday");
  public final static Day[] day = {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
  };
  public final static Day number(int ord) {
    return day[ord - 1];
  }
}

public class E11_DaysOfWeek {
  public static void main(String args[]) {
    Day d = Day.MONDAY;
    System.out.println(d);
    d = Day.number(4);
    System.out.println(d);
    System.out.println(d == Day.SATURDAY);
    System.out.println(d.equals(Day.SATURDAY));
    d = Day.number(6);
    System.out.println(d);
    System.out.println(d == Day.SATURDAY);
    System.out.println(d.equals(Day.SATURDAY));
  }
} ///:~
```

The output is:

```
Monday
Thursday
false
false
Saturday
true
true
```

Exercise 12

```
//: c08:E12_ProtectedInnerClass.java
/****************** Exercise 12 *****************
 * Create an interface with at least one method,
 * in its own package. Create a class in a
 * separate package. Add a protected inner class
 * that implements the interface. In a third
 * package, inherit from your class and, inside a
 * method, return an object of the protected
 * inner class, upcasting to the interface during
 * the return.
 ***********************************************/
import c08.exercise12b.*;
import c08.exercise12.*;

public class E12_ProtectedInnerClass
extends SimpleClass {
  public SimpleInterface get() {
    return new Inner();
  }
  public static void main(String args[]) {
    new E12_ProtectedInnerClass().get().f();
  }
} ///:~
```

```
//: c08:exercise12:SimpleInterface.java
package c08.exercise12;

public interface SimpleInterface {
  void f();
} ///:~
```

```
//: c08:exercise12b:SimpleClass.java
package c08.exercise12b;
import c08.exercise12.*;

public class SimpleClass {
  protected class Inner implements SimpleInterface {
    // Force constructor to be public:
    public Inner() {}
    public void f() {}
  }
} ///:~
```

Note that if you don’t define a constructor and allow the default constructor to be synthesized, it will be given package access and you won’t be able to access it outside the package.

Exercise 13

```
//: c08:E13_InnerClassInMethod.java
/****************** Exercise 13 *****************
 * Create an interface with at least one method,
 * and implement that interface by defining an
 * inner class within a method, which returns a
 * reference to your interface.
 ***********************************************/
import c08.exercise12.*;

public class E13_InnerClassInMethod {
  public SimpleInterface get() {
    class SI implements SimpleInterface{
      public void f() {
        System.out.println("E13_InnerClassInMethod.f");
      }
    }
    return new SI();
  }
  public static void main(String args[]) {
    SimpleInterface si =
      new E13_InnerClassInMethod().get();
    si.f();
  }
} ///:~
```

Exercise 14

```
//: c08:E14_InnerClassInScope.java
/****************** Exercise 14 *****************
 * Repeat Exercise 13 but define the inner class
 * within a scope within a method.
 ***********************************************/
import c08.exercise12.*;

public class E14_InnerClassInScope {
  public SimpleInterface get() {
    {
      class SI implements SimpleInterface{
        public void f() {
          System.out.println("E13_InnerClassInMethod.f");
        }
      }
      return new SI();
    }
  }
  public static void main(String args[]) {
    SimpleInterface si =
      new E13_InnerClassInMethod().get();
    si.f();
  }
} ///:~
```

Note that the **return** statement must be in the same scope as the inner class is defined within, otherwise the inner class will not be visible; the definition of the class goes out of scope.

Exercise 15

```
//: c08:E15_AnonymousInnerClass.java
/****************** Exercise 15 *****************
 * Repeat Exercise 13 using an anonymous inner
 * class.
 ***********************************************/
import c08.exercise12.*;

public class E15_AnonymousInnerClass {
  public SimpleInterface get() {
    return new SimpleInterface() {
      public void f() {
        System.out.println("E15_AnonymousInnerClass.f");
      }
    };
  }
  public static void main(String args[]) {
    SimpleInterface si =
      new E15_AnonymousInnerClass().get();
    si.f();
  }
} ///:~
```

Exercise 16

```
//: c08:E16_HorrorShow.java
/****************** Exercise 16 ******************
 * Modify HorrorShow.java to implement
 * DangerousMonster and Vampire using anonymous
 * classes.
 ***********************************************/

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}
```

```
public class E16_HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DangerousMonster() {
      public void menace() {}
      public void destroy() {}
    };
    u(barney);
    v(barney);
    Vampire vlad = new Vampire() {
      public void menace() {}
      public void destroy() {}
      public void kill() {}
      public void drinkBlood() {}
    };
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

Exercise 17

```
//: c08:E17_HiddenInnerClass.java
/****************** Exercise 17 *****************
 * Create a private inner class that implements a
 * public interface. Write a method that returns
 * a reference to an instance of the private
 * inner class, upcast to the interface. Show
 * that the inner class is completely hidden by
 * trying to downcast to it.
 ***********************************************/
import c08.exercise12.*;
```

```
class Outer {
  private class Inner implements SimpleInterface {
    public void f() {
      System.out.println("Outer.Inner.f");
    }
  }
  public SimpleInterface get() {
    return new Inner();
  }
  public Inner get2() {
    return new Inner();
  }
}

public class E17_HiddenInnerClass {
  public static void main(String args[]) {
    Outer out = new Outer();
    SimpleInterface si = out.get();
    si = out.get2();
    // Won't compile -- 'Inner' not visible:
    //! Inner i1 = out.get2();
    //! Inner i2 = (Inner)si;
  }
} ///:~
```

The **public get( )** method returns an instance of the private class **Inner**, but upcast to the **public SimpleInterface**, as requested by the exercise.

There's an interesting twist with the **get2( )** method: it returns an object of the private class **Inner**, which compiles fine, except that when you call **get2( )** from outside of **Outer**, you can't use the return value's actual type since that type is **private** and not visible outside the class; you can only upcast the return value to the visible base interface. This seems a little strange at first, but it would allow **Outer** methods to receive the actual type while still allowing methods of other classes to call **get2( )** in the more limited way.

Exercise 18

```
//: c08:E18_ReturningAnonymousIC.java
```

```
/****************** Exercise 18 *****************
 * Create a class with a nondefault constructor
 * and no default constructor. Create a second
 * class that has a method which returns a
 * reference to the first class. Create the
 * object to return by making an anonymous inner
 * class that inherits from the first class.
 ***********************************************/
class NoDefault {
  private int i;
  public NoDefault(int i) {
    this.i = i;
  }
  public void f() {
    System.out.println("NoDefault.f");
  }
}

class Second {
  public NoDefault get1(int i) {
    // Doesn't override any methods:
    return new NoDefault(i) {};
  }
  public NoDefault get2(int i) {
    // Overrides f():
    return new NoDefault(i) {
      public void f() {
        System.out.println("Second.get2.f");
      }
    };
  }
}

public class E18_ReturningAnonymousIC {
  public static void main(String args[]) {
    Second sec = new Second();
    NoDefault nd = sec.get1(47);
    nd.f();
    nd = sec.get2(99);
    nd.f();
  }
} ///:~
```

In **get1( )**, **NoDefault** is inherited in the anonymous inner class, but no methods are overriden; you probably won't do this very often. Instead, when you inherit you'll probably override a method as in **get2( )**.

Exercise 19

```
//: c08:E19_InnerClassAccess.java
/****************** Exercise 19 *****************
 * Create a class with a private field and a
 * private method. Create an inner class with a
 * method that modifies the outer-class field and
 * calls the outer-class method. In a second
 * outer-class method, create an object of the
 * inner class and call its method, then show
 * the effect on the outer-class object.
 ***********************************************/
public class E19_InnerClassAccess {
  private int i = 10;
  private void f() {
    System.out.println("E19_InnerClassAccess.f");
  }
  class Inner {
    void g() {
      i++;
      f();
    }
  }
  public void h() {
    Inner in = new Inner();
    in.g();
    System.out.println("i = " + i);
  }
  public static void main(String args[]) {
    E19_InnerClassAccess ica = new E19_InnerClassAccess();
    ica.h();
  }
} ///:~
```

This exercise shows how inner classes have transparent access to their outer-class objects, even their private fields and methods.

The output is:

```
E19_InnerClassAccess.f
i = 11
```

Exercise 20

```
//: c08:E20_AnonymousInnerAccess.java
/****************** Exercise 20 *****************
 * Repeat Exercise 19 using an anonymous inner
 * class.
 ***********************************************/
public class E20_AnonymousInnerAccess {
  private int i = 10;
  private void f() {
    System.out.println(
      "E20_AnonymousInnerAccess.f");
  }
  public void h() {
    new Object() {
      void g() {
        i++;
        f();
      }
    }.g();
    System.out.println("i = " + i);
  }
  public static void main(String args[]) {
    new E20_AnonymousInnerAccess().h();
  }
} ///:~
```

This one is actually a bit tricky – you're told to use an anonymous inner class, but not what to inherit it from. Since an anonymous inner class is always inherited from something, in the absence of any other instructions, we can inherit it from **Object** and then add the **g( )** method, which is then immediately called – this is the only time you can use **g( )**, because otherwise (if you assigned it to an **Object** reference) what would you cast it to in order to call **g( )**? (There may actually be a solution to this, by using Java's *Reflection*, but that won't be covered until later in the book).

Exercise 21

```
//: c08:E21_InnerClass.java
/****************** Exercise 21 *****************
 * Create a class containing a nested class.
 * In main(), create an instance of the inner
 * class.
 ***********************************************/
public class E21_InnerClass {
  static class Inner {
    void f() {
      System.out.println("Inner.f");
    }
  }
  public static void main(String args[]) {
    Inner in = new Inner();
    in.f();
  }
}

class Other {
  // Specifying the inner type outside
  // the scope of the class:
  void f() {
    E21_InnerClass.Inner in = new E21_InnerClass.Inner();
  }
} ///:~
```

Inside a method of the class where the nested (**static** inner) class is defined, you can just refer to it by its class name, but if you're outside of that class, you must fully specify the outer class and inner class, as shown in **Other**, above.

Exercise 22

```
//: c08:E22_InterfaceWithInner.java
```

```
/****************** Exercise 22 *****************
 * Create an interface containing a nested class.
 * Implement this interface and create an
 * instance of the nested class.
 ***********************************************/
interface WithStaticInner {
  static class Inner {
    int i;
    public Inner(int i) {
      this.i = i;
    }
    void f() {
      System.out.println("Inner.f");
    }
  }
}

class B2 implements WithStaticInner {}

public class E22_InterfaceWithInner {
  public static void main(String args[]) {
    B2 b = new B2();
    WithStaticInner.Inner in =
      new WithStaticInner.Inner(5);
    in.f();
  }
} ///:~
```

What's interesting about this is that even though an **interface** can have no implementation, an inner class defined within that **interface** has no such restriction. Thus, **Inner** happens to be defined within **WithStaticInner**, but that just means its name is located there – since all elements of an **interface** are **public**, **Inner** doesn't even have any added access to the elements of **WithStaticInner**.

Exercise 23

```
//: c08:E23_InnerInsideInner.java
/****************** Exercise 23 *****************
 * Create a class containing an inner class that
 * itself contains an inner class. Repeat this
 * using static inner classes. Note the names of
 * the .class files produced by the compiler.
 ***********************************************/
public class E23_InnerInsideInner {
  class Inner1 {
    class Inner2 {
      void f() {}
    }
    Inner2 makeInner2() { return new Inner2(); }
  }
  Inner1 makeInner1() { return new Inner1(); }
  static class Nested1 {
    static class Nested2 {
      void f() {}
    }
    void f() {}
  }
  public static void main(String args[]) {
    new E23_InnerInsideInner.Nested1().f();
    new E23_InnerInsideInner.Nested1.Nested2().f();
    E23_InnerInsideInner x1 = new E23_InnerInsideInner();
    E23_InnerInsideInner.Inner1 x2 = x1.makeInner1();
    E23_InnerInsideInner.Inner1.Inner2 x3 =
      x2.makeInner2();
    x3.f();
  }
} ///:~
```

The class names produced are:

```
E23_InnerInsideInner.class
E23_InnerInsideInner$Inner1.class
E23_InnerInsideInner$Inner1$Inner2.class
E23_InnerInsideInner$Nested1.class
E23_InnerInsideInner$Nested1$Nested2.class
```

Exercise 24

```
//: c08:E24_InstanceOfInnerClass.java
/****************** Exercise 24 *****************
```

```
 * Create a class with an inner class. In a
 * separate class, make an instance of the inner
 * class.
 ***********************************************/
class Class1 {
  class Inner {}
}

public class E24_InstanceOfInnerClass {
  public static void main(String args[]) {
    Class1 c1 = new Class1();
    Class1.Inner ci = c1.new Inner();
  }
} ///:~
```

In the separate class, you must fully resolve the name of the inner class to create a reference, and you must specify the object that the inner class is being created for in the **new** expression (the inner class object has a connection to the outer-class object).

Exercise 25

```
//: c08:E25_InnerClassInheritance.java
/****************** Exercise 25 *****************
 * Create a class with an inner class that has a
 * nondefault constructor. Create a second class
 * with an inner class that inherits from the
 * first inner class.
 ***********************************************/
class WithNonDefault {
  class Inner {
    int i;
    public Inner(int i) { this.i = i; }
    public Inner() { i = 47; }
    public void f() { System.out.println("Inner.f"); }
  }
}

public class E25_InnerClassInheritance {
  class Inner2 extends WithNonDefault.Inner {
    // Won't compile -- WithNonDefault not available:
    //! public Inner2(int i) {
    //!   super(i);
    //! }
    public Inner2(WithNonDefault wnd, int i) {
      wnd.super(i);
    }
    public void f() {
      System.out.println("Inner2.f");
      super.f();
    }
  }
  public static void main(String args[]) {
    WithNonDefault wnd = new WithNonDefault();
    E25_InnerClassInheritance ici =
      new E25_InnerClassInheritance();
    Inner2 i2 = ici.new Inner2(wnd, 47);
    i2.f();
  }
} ///:~
```

To create an instance of an inner class, you need to use the **new** expression involving the outer-class object. And to create an instance of an inner class inheriting from another inner class, you must provide the constructor with an instance of the outer base class. The creation of a new **Inner2** object thus becomes doubly complex. Note the result, however: an **Inner2** object has an intimate connection with the **WithNonDefault** object and with the **E25_InnerClassInheritance** object, so it may be a way to implement a very private mediation between the two objects (see *Mediator* in design patterns literature).

Exercise 26

```
//: c08:E26_RepairWindError.java
/****************** Exercise 26 *****************
 * Repair the problem in WindError.java.
 ***********************************************/
class NoteX {
  public static final int
    MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX {
  public void play(int NoteX) {
    System.out.println("InstrumentX.play()");
  }
}

class WindX extends InstrumentX {
  // OOPS! Changes the method interface:
  public void play(int NoteX) {
    System.out.println("WindX.play(int NoteX)");
  }
}

class WindError {
  public static void tune(InstrumentX i) {
    // ...
    i.play(NoteX.MIDDLE_C);
  }
}


public class E26_RepairWindError {
  public static void main(String[] args) {
    WindX flute = new WindX();
    WindError.tune(flute);
  }
} ///:~
```

The output is now:

```
WindX.play(int NoteX)
```

Exercise 27

```
//: c08:E27_GetRSelector.java
/****************** Exercise 27 *****************
 * Modify Sequence.java by adding a method
 * getRSelector() that produces a different
 * implementation of the Selector interface that
 * moves backward through the sequence from the
 * end to the beginning.
 ***********************************************/
interface Selector {
  boolean end();
  Object current();
  void next();
}

class Sequence {
  private Object[] obs;
  private int next = 0;
  public Sequence(int size) {
    obs = new Object[size];
  }
  public void add(Object x) {
    if(next < obs.length) {
      obs[next] = x;
      next++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean end() {
      return i == obs.length;
    }
    public Object current() {
      return obs[i];
    }
    public void next() {
      if(i < obs.length) i++;
    }
  }
  private class RSelector implements Selector {
    int i = obs.length - 1;
    public boolean end() {
      return i < 0;
    }
    public Object current() {
      return obs[i];
    }
    public void next() {
      if(i >= 0) i--;
    }
  }
  public Selector getSelector() {
    return new SSelector();
  }
  public Selector getRSelector() {
    return new RSelector();
  }
}

public class E27_GetRSelector {
  public static void main(String[] args) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    while(!sl.end()) {
      System.out.println(sl.current());
      sl.next();
    }
    System.out.println("************");
    Selector sl2 = s.getRSelector();
    while(!sl2.end()) {
      System.out.println(sl2.current());
      sl2.next();
    }
  }
} ///:~
```

The solution is mostly copy and paste, with some minor logic changes in the **RSelector** class. The output is:

```
0
1
2
3
4
5
6
7
```

```
8
9
************
9
8
7
6
5
4
3
2
1
0
```

Exercise 28

```
//: c08:E28_UAB.java
/****************** Exercise 28 *****************
 * Create an interface U with three methods.
 * Create a class A with a method that produces a
 * reference to a U by building an anonymous
 * inner class. Create a second class B that
 * contains an array of U. B should have one
 * method that accepts and stores a reference to
 * a U in the array, a second method that sets a
 * reference in the array (specified by the
 * method argument) to null, and a third method
 * that moves through the array and calls the
 * methods in U. In main(), create a group of A
 * objects and a single B. Fill the B with U
 * references produced by the A objects. Use the
 * B to call back into all the A objects. Remove
 * some of the U references from the B.
 ***********************************************/

interface U {
  void f();
  void g();
  void h();
}
```

```
class A {
  public U getU() {
    return new U() {
      public void f() {
        System.out.println("A.f");
      }
      public void g() {
        System.out.println("A.g");
      }
      public void h() {
        System.out.println("A.h");
      }
    };
  }
}

class B {
  U[] ua;
  public B(int size) {
    ua = new U[size];
  }
  public boolean add(U elem) {
    for(int i = 0; i < ua.length; i++) {
      if(ua[i] == null) {
        ua[i] = elem;
        return true;
      }
    }
    return false; // Couldn't find any space
  }
  public boolean setNull(int i) {
    if(i < 0 || i >= ua.length)
      return false; // Value out of bounds
    // (Normally throw an exception)
    ua[i] = null;
    return true;
  }
  public void callMethods() {
    for(int i = 0; i < ua.length; i++)
      if(ua[i] != null) {
        ua[i].f();
        ua[i].g();
        ua[i].h();
      }
  }
}

public class E28_UAB {
  public static void main(String args[]) {
    A[] aa = { new A(), new A(), new A() };
    B b = new B(3);
    for(int i = 0; i < aa.length; i++)
      b.add(aa[i].getU());
    b.callMethods();
    System.out.println("****");
    b.setNull(0);
    b.callMethods();
  }
} ///:~
```

The output is:

```
A.f
A.g
A.h
A.f
A.g
A.h
A.f
A.g
A.h
****
A.f
A.g
A.h
A.f
A.g
A.h
```

Here you can see that the zeroth element was removed.

Exercise 29

```
//: c08:E29_GreenhouseInnerEvent.java
/****************** Exercise 29 *****************
 * In GreenhouseControls.java, add Event inner
 * classes that turn fans on and off. Configure
 * GreenhouseController.java to use these new
 * Event objects.
 ***********************************************/
import java.util.*;

abstract class Event {
  private long eventTime;
  protected final long delayTime;
  public Event(long delayTime) {
    this.delayTime = delayTime;
    start();
  }
  public void start() { // Allows restarting
    eventTime = System.currentTimeMillis() + delayTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= eventTime;
  }
  public abstract void action();
}

class Controller {
  // An object from java.util to hold Event objects:
  private List eventList = new ArrayList();
  public void addEvent(Event c) { eventList.add(c); }
  public void run() {
    while(eventList.size() > 0) {
      for(int i = 0; i < eventList.size(); i++) {
        Event e = (Event)eventList.get(i);
        if(e.ready()) {
          System.out.println(e);
          e.action();
          eventList.remove(i);
        }
      }
    }
  }
}

class GreenhouseControls extends Controller {
  private boolean fan = false;
  private boolean light = false;
  public class FanOn extends Event {
    public FanOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the Fan.
      fan = true;
    }
    public String toString() { return "Fan is on"; }
  }
  public class FanOff extends Event {
    public FanOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the Fan.
      fan = false;
    }
    public String toString() { return "Fan is off"; }
  }
  public class LightOn extends Event {
    public LightOn(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String toString() { return "Light is on"; }
  }
  public class LightOff extends Event {
    public LightOff(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String toString() { return "Light is off"; }
  }
  private boolean water = false;
  public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here.
      water = true;
    }
    public String toString() {
      return "Greenhouse water is on";
    }
  }
  public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here.
      water = false;
    }
    public String toString() {
      return "Greenhouse water is off";
    }
  }
  private String thermostat = "Day";
  public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here.
      thermostat = "Night";
    }
    public String toString() {
      return "Thermostat on night setting";
    }
  }
```

```
public class ThermostatDay extends Event {
  public ThermostatDay(long delayTime) {
    super(delayTime);
  }
  public void action() {
    // Put hardware control code here.
    thermostat = "Day";
  }
  public String toString() {
    return "Thermostat on day setting";
  }
}
// An example of an action() that inserts a
// new one of itself into the event list:
public class Bell extends Event {
  public Bell(long delayTime) { super(delayTime); }
  public void action() {
    addEvent(new Bell(delayTime));
  }
  public String toString() { return "Bing!"; }
}
public class Restart extends Event {
  private Event[] eventList;
  public Restart(long delayTime, Event[] eventList) {
    super(delayTime);
    this.eventList = eventList;
    for(int i = 0; i < eventList.length; i++)
      addEvent(eventList[i]);
  }
  public void action() {
    for(int i = 0; i < eventList.length; i++) {
      eventList[i].start(); // Rerun each event
      addEvent(eventList[i]);
    }
    start(); // Rerun this Event
    addEvent(this);
  }
  public String toString() {
    return "Restarting system";
  }
}
```

```
  public class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Terminating";  }
  }
}

public class E29_GreenhouseInnerEvent {
  public static void main(String[] args) {
    GreenhouseControls gc = new GreenhouseControls();
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
      gc.new LightOn(200),
      gc.new FanOn(300),
      gc.new LightOff(400),
      gc.new FanOff(500),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    gc.addEvent(gc.new Terminate(8000));
    gc.run();
  }
} ///:~
```

This is basically a copy and paste exercise, but it helps ensure that you understand the structure of the program.

Exercise 30

```
//: c08:E30_GreenhouseControls.java
/****************** Exercise 30 ******************
 * Inherit from GreenhouseControls in
 * GreenhouseControls.java to add Event inner
 * classes that turn water mist generators on
 * and off. Write a new version of
 * GreenhouseController.java to use these new
 * Event objects.
 ***********************************************/
```

```
public class E30_GreenhouseControls
  extends GreenhouseControls {
  private boolean generator = false;
  public class WatermistGeneratorOn extends Event {
    public WatermistGeneratorOn(long delayTime) {
      super(delayTime);
    }
    public void action() {
      generator = true;
    }
    public String toString() {
      return "Water mist generator is on";
    }
  }
  public class WatermistGeneratorOff extends Event {
    public WatermistGeneratorOff(long delayTime) {
      super(delayTime);
    }
    public void action() {
      generator = false;
    }
    public String toString() {
      return "Water mist generator is off";
    }
  }
} ///:~
```

```
//: c08:E30_GreenhouseController.java
// {Args: 5000}

public class E30_GreenhouseController {
  public static void main(String[] args) {
    E30_GreenhouseControls gc =
      new E30_GreenhouseControls();
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
      gc.new LightOn(200),
      gc.new LightOff(400),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400),
      gc.new WatermistGeneratorOn(1600),
      gc.new WatermistGeneratorOff(1800)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    gc.addEvent(gc.new Terminate(8000));
    gc.run();
  }
} ///:~
```

Exercise 31

```
//: c08:E31_AccessibilityInnerClasses.java
/****************** Exercise 31 *****************
 * Show that an inner class has access to the
 * private elements of its outer class. Determine
 * whether the reverse is true.
 ***********************************************/
class X {
  private int i;
  private void f(String s) {
    System.out.println("X.f");
    System.out.println("s = " + s);
  }
  class Inner {
    private int j;
    private void h() {
      System.out.println("Inner.h");
    }
    public void g() {
      i = 99;
      f("Called in Inner");
    }
  }
  public void testInnerAccess() {
    Inner i = new Inner();
    i.h();
    i.j = 47;
  }
}

public class E31_AccessibilityInnerClasses {
  public static void main(String args[]) {
    X x = new X();
    X.Inner xi = x.new Inner();
    xi.g();
    x.testInnerAccess();
  }
} ///:~
```

As you can see from the above example, the accessibility does in fact go both ways. The output is:

```
X.f
s = Called in Inner
Inner.h
```

Chapter 9

Exercise 1

```
//: c09:E01_SimpleException.java
/****************** Exercise 1 ******************
 * Create a class with a main() that throws an
 * object of class Exception inside a try block.
 * Give the constructor for Exception a String
 * argument. Catch the exception inside a catch
 * clause and print the String argument. Add a
 * finally clause and print a message to prove
 * you were there.
 ***********************************************/
public class E01_SimpleException {
  public static void main(String args[]) {
    try {
      throw new Exception("An exception in main");
    } catch(Exception e) {
      System.err.println(
        "e.getMessage() = " + e.getMessage());
    } finally {
      System.out.println("In finally clause");
    }
  }
} ///:~
```

The output is:

```
e.getMessage() = An exception in main
In finally clause
```

Exercise 2

```
//: c09:E02_ExceptionClass.java
/****************** Exercise 2 ******************
 * Create your own exception class using the
 * extends keyword. Write a constructor for this
 * class that takes a String argument and stores
 * it inside the object with a String reference.
 * Write a method that prints out the stored
 * String. Create a try-catch clause to exercise
 * your new exception.
 ***********************************************/
// Following the instructions to the letter:
class MyException extends Exception {
  String msg;
  public MyException(String msg) {
    this.msg = msg;
  }
  public void printMsg() {
    System.err.println("msg = " + msg);
  }
}

// Or you can take a more clever approach, and
// note that string storage and printing is
// built into Exception:
class MyException2 extends Exception {
  public MyException2(String s) { super(s); }
}

public class E02_ExceptionClass {
  public static void main(String args[]) {
    try {
      throw new MyException("MyException message");
    } catch(MyException e) {
      e.printMsg();
    }
    try {
      throw new MyException2("MyException2 message");
    } catch(MyException2 e) {
      System.err.println(
        "e.getMessage() = " + e.getMessage());
    }
  }
} ///:~
```

Here's the output:

```
msg = MyException message
e.getMessage() = MyException2 message
```

Exercise 3

```
//: c09:E03_ExceptionSpecification.java
/****************** Exercise 3 ******************
 * Write a class with a method that throws an
 * exception of the type created in Exercise 2.
 * Try compiling it without an exception
 * specification to see what the compiler says.
 * Add the appropriate exception specification.
 * Try out your class and its exception inside a
 * try-catch clause.
 ***********************************************/
class Thrower {
  public void f() {
    // Compiler gives an error: "unreported
    // exception MyException; must be caught or
    // declared to be thrown"
    //! throw new MyException("Inside f()");
  }
  public void g() throws MyException {
    throw new MyException("Inside f()");
  }
}

public class E03_ExceptionSpecification {
  public static void main(String args[]) {
    Thrower t = new Thrower();
    try {
      t.g();
    } catch(MyException e) {
      e.printMsg();
    }
  }
} ///:~
```

The error message given above is from IBM's free "jikes" compiler, which tends to give more illuminating error messages than Sun's JDK.

The output is:

```
msg = Inside f()
```

Exercise 4

```
//: c09:E04_NullReference.java
/****************** Exercise 4 ******************
 * Define an object reference and initialize it
 * to null. Try to call a method through this
 * reference. Now wrap the code in a try-catch
 * clause to catch the exception.
 ***********************************************/
public class E04_NullReference {
  public static void main(String args[]) {
    String s = null;
    // Causes a NullPointerException:
    //! s.toString();
    try {
      s.toString();
    } catch(Exception e) {
      System.err.println("Caught exception " + e);
    }
  }
} ///:~
```

When you catch the exception, the program runs to completion. The output is:

```
Caught exception java.lang.NullPointerException
```

Exercise 5

```
//: c09:E05_ChangeException.java
/****************** Exercise 5 ******************
 * Create a class with two methods, f() and g().
 * In g(), throw an exception of a new type that
 * you define. In f(), call g(), catch its
 * exception and, in the catch clause, throw a
 * different exception (of a second type that you
 * define). Test your code in main().
 ***********************************************/
class AnException extends Exception {}

class AnotherException extends Exception {}

public class E05_ChangeException {
  public void g() throws AnException {
    throw new AnException();
  }
  public void f() throws AnotherException {
    try {
      g();
    } catch(AnException e) {
      throw new AnotherException();
    }
  }
  public static void main(String args[]) {
    E05_ChangeException ce = new E05_ChangeException();
    try {
      ce.f();
    } catch(AnotherException e) {
      System.err.println("Caught " + e);
    }
  }
} ///:~
```

Once the exception is caught in a **catch** clause, it's considered handled and you can do anything you want, including throw another exception, as is done here. The output is:

```
Caught AnotherException
```

Exercise 6

```
//: c09:E06_ChangeToRuntimeException.java
// {ThrowsException}
/****************** Exercise 6 ******************
```

```
 * Repeat the previous exercise, but inside the
 * catch clause, wrap g()'s exception in a
 * RuntimeException.
 ***********************************************/

class AnException2 extends Exception {}

public class E06_ChangeToRuntimeException {
  public void g() throws AnException2 {
    throw new AnException2();
  }
  public void f() {
    try {
      g();
    } catch(AnException2 e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String args[]) {
    E06_ChangeToRuntimeException ce =
      new E06_ChangeToRuntimeException();
    ce.f();
  }
} ///:~
```

Since **Exception** is now wrapped in a **RuntimeException** we do not need to use a try block.

Exercise 7

```
//: c09:E07_CatchAll.java
/****************** Exercise 7 ******************
 * Create three new types of exceptions. Write a
 * class with a method that throws all three. In
 * main(), call the method but only use a single
 * catch clause that will catch all three types
 * of exceptions.
 ***********************************************/
class ExBase extends Exception {}
class Ex1 extends ExBase {}
class Ex2 extends ExBase {}
class Ex3 extends ExBase {}

class Thrower2 {
  void f() throws Ex1, Ex2, Ex3 {
    throw new Ex1();
    // You aren't forced to throw all the
    // exceptions in the specification.
  }
}

public class E07_CatchAll {
  public static void main(String args[]) {
    Thrower2 t = new Thrower2();
    try {
      t.f();
    } catch(ExBase e) {
      System.err.println("caught " + e);
    } catch(Exception e) {
      System.err.println("caught " + e);
    }
  }
} ///:~
```

The approach shown here creates a common base class for all three exceptions, and then just catches the common base exception. However, you can also observe that all exceptions are inherited from **Exception** and just catch that.

Exercise 8

```
//: c09:E08_ArrayIndexBounds.java
/****************** Exercise 8 ******************
 * Write code to generate and catch an
 * ArrayIndexOutOfBoundsException.
 ***********************************************/
public class E08_ArrayIndexBounds {
  public static void main(String args[]) {
    char[] array = new char[10];
    try {
      array[10] = 'x';
    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("e = " + e);
    }
  }
} ///:~
```

The output is:

```
e = java.lang.ArrayIndexOutOfBoundsException
```

Exercise 9

```
//: c09:E09_Resumption.java
/****************** Exercise 9 ******************
 * Create your own resumption-like behavior using
 * a while loop that repeats until an exception
 * is no longer thrown.
 ***********************************************/
class ResumerException extends Exception {}

class Resumer {
  static int count = 3;
  void f() throws ResumerException {
    if(--count > 0)
      throw new ResumerException();
  }
}

public class E09_Resumption {
  public static void main(String args[]) {
    Resumer r = new Resumer();
    while(true) {
      try {
        r.f();
      } catch(ResumerException e) {
        System.err.println("Caught " + e);
        continue;
      }
      System.out.println("Got through...");
      break;
    }
    System.out.println("Successful execution");
  }
} ///:~
```

There are probably other approaches. The point is that as long as an exception is thrown, you go back and try again – presumably after trying to fix the problem.

The output is:

```
Caught ResumerException
Caught ResumerException
Got through...
Successful execution
```

Exercise 10

```
//: c09:E10_ThreeLevelExceptions.java
/****************** Exercise 10 ******************
 * Create a three-level hierarchy of exceptions.
 * Now create a base class A with a method that
 * throws an exception at the base of your
 * hierarchy. Inherit B from A and override the
 * method so it throws an exception at level two
 * of your hierarchy. Repeat by inheriting class
 * C from B. In main(), create a C and upcast it
 * to A, then call the method.
 ***********************************************/
class Level1Exception extends Exception {}
class Level2Exception extends Level1Exception {}
class Level3Exception extends Level2Exception {}

class A {
  public void f() throws Level1Exception {
    throw new Level1Exception();
  }
}

class B extends A {
  public void f() throws Level2Exception {
    throw new Level2Exception();
  }
}

class C extends B {
  public void f() throws Level3Exception {
    throw new Level3Exception();
  }
}

public class E10_ThreeLevelExceptions {
  public static void main(String args[]) {
    A a = new C();
    try {
      a.f();
    } catch(Level1Exception e) {
      System.err.println("Caught " + e);
    }
  }
} ///:~
```

This is a fairly connect-the-dots solution – you just follow the instructions. The output is:

```
Caught Level3Exception
```

The compiler forces you to catch a **Level1Exception** because that's what **A.f( )** throws.

Exercise 11

```
//: c09:E11_ConstructorExceptions.java
/****************** Exercise 11 *****************
 * Demonstrate that a derived-class constructor
 * cannot catch exceptions thrown by its
 * base-class constructor.
 ***********************************************/
class Except1 extends Exception {
  public Except1(String s) {
    super(s);
  }
}
```

```
class BaseWithException {
  public BaseWithException() throws Except1 {
    throw new Except1("thrown by BaseWithException");
  }
}

class DerivedWE extends BaseWithException {
  // Produces compile-time error:
  //    unreported exception Except1
  // ! public DerivedWE() {}
  // Gives compile error: call to super must be
  // first statement in constructor:
  //! public DerivedWE() {
  //!   try {
  //!     super();
  //!   } catch(Except1 ex1) {
  //!   }
  //! }
  public DerivedWE() throws Except1 {}
}

public class E11_ConstructorExceptions {
  public static void main(String args[]) {
    try {
      new DerivedWE();
    } catch(Except1 ex1) {
      System.err.println("Caught " + ex1);
    }
  }
} ///:~
```

The output is:

```
Caught Except1: thrown by BaseWithException
```

Exercise 12

```
//: c09:E12_OnOffSwitch.java
/****************** Exercise 12 *****************
 * Show that OnOffSwitch.java can fail by
 * throwing a RuntimeException inside the try
 * block.
 *********************************************/
class Switch {
  boolean state = false;
  boolean read() { return state; }
  void on() { state = true; }
  void off() { state = false; }
  public String toString() {
    return "Switch = " + (state ? "on" : "off");
  }
}

class OnOffException1 extends Exception {}
class OnOffException2 extends Exception {}

public class E12_OnOffSwitch {
  static Switch sw = new Switch();
  static void f() throws OnOffException1, OnOffException2 {
    throw new RuntimeException("Inside try");
  }
  public static void main(String[] args) {
    try {
      try {
        sw.on();
        // Code that can throw exceptions...
        f();
        sw.off();
      } catch(OnOffException1 e) {
        System.err.println("OnOffException1");
        sw.off();
      } catch(OnOffException2 e) {
        System.err.println("OnOffException2");
        sw.off();
      }
    } catch(RuntimeException e) {
      System.err.println(sw);
      System.err.println("Oops! the exception '"
        + e + "' slipped through without "
        + "turning the switch off!");
    }
  }
} ///:~
```

The output is:

```
Switch = on
Oops! the exception 'java.lang.RuntimeException:
Inside try' slipped through without turning the
switch off!
```

Exercise 13

```
//: c09:E13_WithFinally.java
/****************** Exercise 13 *****************
 * Show that WithFinally.java doesn't fail by
 * throwing a RuntimeException inside the try
 * block.
 ***********************************************/
public class E13_WithFinally {
  static Switch sw = new Switch();
  static void f() throws
    OnOffException1, OnOffException2 {
    throw new RuntimeException("Inside try");
  }
  public static void main(String[] args) {
    try {
      try {
        sw.on();
        // Code that can throw exceptions...
        f();
      } catch(OnOffException1 e) {
        System.err.println("OnOffException1");
      } catch(OnOffException2 e) {
        System.err.println("OnOffException2");
      } finally {
        sw.off();
      }
    } catch(RuntimeException e) {
      System.err.println("Exception '" + e +
        "'. Did the switch get turned off?");
      System.err.println(sw);
    }
  }
} ///:~
```

The output is:

```
Exception 'java.lang.RuntimeException:
Inside try'. Did the switch get turned off?
Switch = off
```

You can see that the finally clause guarantees that any necessary cleanup occurs, even with an unexpected exception.

Exercise 14

```
//: c09:E14_Finally.java
// {ThrowsException}
/****************** Exercise 14 *****************
 * Modify Exercise 7 by adding a finally clause.
 * Verify that your finally clause is executed, even
 * if a NullPointerException is thrown.
 ***********************************************/
public class E14_Finally {
  public static void throwNull() {
    throw new NullPointerException();
  }
  public static void main(String args[]) {
    Thrower2 t = new Thrower2();
    try {
      t.f();
    } catch(ExBase e) {
      System.err.println("caught " + e);
    } finally {
      System.out.println("In finally clause A");
    }
    try {
      throwNull();
      t.f();
    } catch(ExBase e) {
      System.err.println("caught " + e);
    } finally {
      System.out.println("In finally clause B");
    }
  }
} ///:~
```

The output is:

```
caught Ex1
In finally clause A
In finally clause B
Exception in thread "main" java.lang.NullPointerException
        at E14_Finally.throwNull(E14_Finally.java:10)
        at E14_Finally.main(E14_Finally.java:22)
```

You can see that even though the **NullPointerException** was not caught, **finally** clause B was still executed.

Exercise 15

```
//: c09:E15_DisposeFlag.java
/****************** Exercise 15 *****************
 * Create an example where you use a flag to
 * control whether cleanup code is called, as
 * described in the second paragraph after the
 * heading "Constructors."
 ***********************************************/
// The paragraph reads:
/*
Since you've just learned about finally, you
might think that it is the correct solution. But
it's not quite that simple, because finally
performs the cleanup code every time, even in
the situations in which you don't want the
cleanup code executed until the cleanup method
runs. Thus, if you do perform cleanup in finally,
you must set some kind of flag when the
constructor finishes normally so that you don't
do anything in the finally block if the flag is
set. Because this isn't particularly elegant (you
are coupling your code from one place to
another), it's best if you try to avoid
performing this kind of cleanup in finally
```

```
unless you are forced to.
*/
class AbortedConstruction extends Exception {
  public AbortedConstruction() {
    super("Construction aborted");
  }
}

class WithDispose {
  private boolean constructed = false;
  public WithDispose(boolean abort)
    throws AbortedConstruction {
    // Perform construction that might be
    // unsucessful (and throw an exception) here.
    if(abort) throw new AbortedConstruction();
    System.out.println("After exception");
    constructed = true;
  }
  public void dispose() {
    System.out.println("constructed = " + constructed);
    if(constructed == true)
      System.out.println("Cleaning up");
    else
      System.out.println("Constructor didn't finish," +
        " not cleaning up");
  }
}

public class E15_DisposeFlag {
  public static void main(String args[]) {
    WithDispose wc = null;
    try {
      wc = new WithDispose(false);
    } catch(AbortedConstruction e) {
      System.err.println("Caught " + e);
    } finally {
      System.out.println(
        "In finally 1, preparing to clean up");
      wc.dispose();
    }
    wc = null; // Very important!
    try {
      try {
        wc = new WithDispose(true);
      } catch(AbortedConstruction e) {
        System.err.println("Caught " + e);
      } finally {
        System.out.println(
          "In finally 2, preparing to clean up");
        wc.dispose();
      }
    } catch(Exception e) {
      System.err.println("Caught exception "+ e);
    }
  }
} ///:~
```

This exercise is particularly fascinating. First of all, in **main( )** note the line commented “very important.” Without this, **wc** remains connected to the object that it was (sucessfully) initialized to in the first **try** block and so in the **finally** clause it appears that **dispose( )** is working with a successfully-created object, or that somehow the **constructed** flag was incorrectly set in the unsuccessfully-created object. Try commenting out the **wc = null** line and you’ll see this behavior.

Probably the most interesting thing about this example is that it shows how *no object at all* is created if an exception is thrown. It also explains why a derived-class constructor cannot catch a base-class constructor exception – it makes no sense to try to “recover” from such an exception failure, since there’s no base-class subobject as a result of the exception.

The reason I needed to put the last **try** inside a second **try** block was so that the build for this solution guide would finish successfully. In the finally clause, when **wc.dispose( )** is called, it encounters a null pointer and throws an exception, which is caught in the outer **try-catch** clause.

The output is:

```
After exception
In finally 1, preparing to clean up
constructed = true
Cleaning up
```

```
Caught AbortedConstruction: Construction aborted
In finally 2, preparing to clean up
Caught exception java.lang.NullPointerException
```

Exercise 16

```
//: c09:E16_UmpireArgument.java
/****************** Exercise 16 *****************
 * Modify StormyInning.java by adding an
 * UmpireArgument exception type and methods
 * that throw this exception. Test the modified
 * hierarchy.
 ***********************************************/
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}
class UmpireArgument extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {}
  abstract void atBat()
    throws Strike, Foul, UmpireArgument;
  abstract void decision() throws UmpireArgument;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}

class StormyInning extends Inning
  implements Storm {
  StormyInning() throws RainedOut,
    BaseballException {}
  StormyInning(String s) throws Foul,
    BaseballException {}
  public void rainHard() throws RainedOut {}
  public void event() {}
  void atBat() throws PopFoul, UmpireArgument {
    throw new UmpireArgument();
  }
  void decision() throws UmpireArgument {
    throw new UmpireArgument();
  }
}

public class E16_UmpireArgument {
  public static void main(String[] args) {
    // Same code as before, still catches
    // the new exception:
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.err.println("Pop foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
      System.err.println("Generic error");
    }
    // Strike not thrown in derived version.
    try {
      Inning i = new StormyInning();
      i.atBat();
    } catch(Strike e) {
      System.err.println("Strike");
    } catch(Foul e) {
      System.err.println("Foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
      System.err.println("Generic baseball exception");
    }
    // Or you can add code to catch the
    // specific type of exception:
```

```
    try {
      StormyInning si = new StormyInning();
      si.atBat();
      si.decision();
    } catch(PopFoul e) {
      System.err.println("Pop foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(UmpireArgument e) {
      System.err.println(
        "Argument with the umpire");
    } catch(BaseballException e) {
      System.err.println("Generic error");
    }
  }
} ///:~
```

The output is:

```
Generic error
Generic baseball exception
Argument with the umpire
```

The first two lines show that by using an exception hierarchy you allow for new exceptions to be added without forcing changes to existing code.

Exercise 17

```
//: c09:E17_Human.java
/****************** Exercise 17 *****************
 * Remove the first catch clause in Human.java
 * and verify that the code still compiles and
 * runs properly.
 ***********************************************/
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class E17_Human {
  public static void main(String[] args) {
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
  }
} ///:~
```

This works because **Annoyance** is the base class for **Sneeze**, so when **Sneeze** is thrown, an **Annoyance** catch statement can catch it. The output is:

```
Caught Annoyance
```

Exercise 18

```
//: c09:E18_LostMessage.java
/****************** Exercise 18 *****************
 * Add a second level of exception loss to
 * LostMessage.java so that the HoHumException is
 * itself replaced by a third exception.
 ***********************************************/
class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}

class YetAnotherException extends Exception {
  public String toString() {
    return "Yet another exception";
  }
}

class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  void cleanup() throws YetAnotherException {
    throw new YetAnotherException();
  }
}

public class E18_LostMessage {
  public static void main(String[] args)
  throws Exception {
    LostMessage lm = new LostMessage();
    try {
      try {
        try {
          lm.f();
        } finally {
          lm.dispose();
        }
      } finally {
        lm.cleanup();
      }
    } catch(YetAnotherException e) {
      System.err.println("Caught " + e);
    }
  }
} ///:~
```

Again, the outer **try** block is so that the program can successfully run to completion, but it also points out that the exception that finally appears is not **VeryImportantException** nor **HoHumException** but rather **YetAnotherException**.

Exercise 19

```
//: c09:E19_GreenhouseExceptions.java
/****************** Exercise 19 *****************
 * Add an appropriate set of exceptions to
 * c08:GreenhouseControls.java.
 *********************************************/
import java.util.*;

// You can also use checked exceptions:
class ActionFailed extends RuntimeException {
  public ActionFailed(String s) {
    super(s);
  }
}

abstract class Event {
  private long eventTime;
  protected final long delayTime;
  public Event(long delayTime) {
    this.delayTime = delayTime;
    start();
  }
  public void start() { // Allows restarting
    eventTime = System.currentTimeMillis() + delayTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= eventTime;
  }
  public abstract void action();
}

class Controller {
  // An object from java.util to hold Event objects:
  private List eventList = new ArrayList();
  public void addEvent(Event c) { eventList.add(c); }
  public void run() {
    while(eventList.size() > 0) {
      for(int i = 0; i < eventList.size(); i++) {
        Event e = (Event)eventList.get(i);
        if(e.ready()) {
          System.out.println(e);
          e.action();
          eventList.remove(i);
        }
      }
    }
  }
}

class GreenhouseControls extends Controller {
  private boolean light = false;
  public class LightOn extends Event {
    public LightOn(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      if(light)
        throw new ActionFailed("Light already on");
      light = true;
    }
    public String toString() { return "Light is on"; }
  }
  public class LightOff extends Event {
    public LightOff(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      if(!light)
        throw new ActionFailed("Light already off");
      light = false;
    }
    public String toString() { return "Light is off"; }
  }
  private boolean water = false;
  public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here.
      if(water)
        throw new ActionFailed("Water already on");
      water = true;
    }
    public String toString() {
      return "Greenhouse water is on";
    }
  }
  public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here.
      if(!water)
        throw new ActionFailed("Water already off");
      water = false;
    }
    public String toString() {
      return "Greenhouse water is off";
    }
  }
  private String thermostat = "Day";
  public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here.
      if(thermostat.equals("Night"))
        throw new ActionFailed(
          "Thermostat already on night setting");
      thermostat = "Night";
    }
    public String toString() {
      return "Thermostat on night setting";
    }
  }
  public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here.
      if(thermostat.equals("Day"))
        throw new ActionFailed(
          "Thermostat already on day setting");
      thermostat = "Day";
    }
    public String toString() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
      addEvent(new Bell(delayTime));
    }
    public String toString() { return "Bing!"; }
  }
  public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
      super(delayTime);
      this.eventList = eventList;
      for(int i = 0; i < eventList.length; i++)
        addEvent(eventList[i]);
    }
    public void action() {
      for(int i = 0; i < eventList.length; i++) {
        eventList[i].start(); // Rerun each event
        addEvent(eventList[i]);
      }
      start(); // Rerun this Event
      addEvent(this);
    }
    public String toString() {
      return "Restarting system";
    }
  }
  public class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Terminating";  }
  }
}

public class E19_GreenhouseExceptions {
  public static void main(String[] args) {
    GreenhouseControls gc = new GreenhouseControls();
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
      gc.new LightOn(200),
      gc.new LightOff(400),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    gc.addEvent(gc.new Terminate(8000));
    gc.run();
  }
} ///:~
```

Of course, this is only one way to approach the problem. You could also have created an exception hierarchy.

Exercise 20

```
//: c09:E20_SequenceExceptions.java
// {ThrowsException}
/************************ Exercise 20 *******************
 * Add an appropriate set of exceptions to
 * c08:Sequence.java.
 ******************************************************/
class SequenceFullException extends RuntimeException {}

class EndOfSequenceException extends RuntimeException {}

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class E20_SequenceExceptions {
  private Object[] objects;
  private int next = 0;
  public E20_SequenceExceptions(int size) {
    objects = new Object[size];
  }
  public void add(Object x) {
    if(next >= objects.length)
      throw new SequenceFullException();
    objects[next++] = x;
  }
  private class SSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == objects.length; }
    public Object current() { return objects[i]; }
    public void next() {
      if(i < objects.length)
        i++;
      else
        throw new EndOfSequenceException();
    }
  }
  public Selector getSelector() { return new SSelector(); }
  public static void main(String[] args) {
    E20_SequenceExceptions sequence =
      new E20_SequenceExceptions(10);
    try {
      for(int i = 0; i < 11; i++)
        sequence.add(Integer.toString(i));
    } catch(SequenceFullException e) {
      System.err.println(e);
    }
    Selector selector = sequence.getSelector();
    while(!selector.end()) {
      System.out.println(selector.current());
      selector.next();
    }
    selector.next();
  }
} ///:~
```

These are more appropriate as **RuntimeException**s because they indicate programmer errors. The output is:

```
SequenceFullException
0
1
2
```

```
3
4
5
6
7
8
9
Exception in thread "main" EndOfSequenceException
        at
E20_SequenceExceptions$SSelector.next(E20_SequenceException
s.java:36)

        at
E20_SequenceExceptions.main(E20_SequenceExceptions.java:54)
```

Exercise 21

```
//: c09:E21_MainException2.java
// {ThrowsException}
/****************** Exercise 21 ******************
 * Change the file name string in MainException.java to
 * name a file that doesn't exist. Run the program and
 * note the result
 ***********************************************/
import java.io.*;

public class E21_MainException2 {
  // Pass all exceptions to the console:
  public static void main(String[] args) throws Exception {
    // Open the file:
    FileInputStream file =
      new FileInputStream("DoesnotExist.file");
    // Use the file ...
    // Close the file:
    file.close();
  }
} ///:~
```

The output is:

```
Exception in thread "main" java.io.FileNotFoundException:
DoesnotExist.file (The
 system cannot find the file specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at
E21_MainException2.main(E21_MainException2.java:14)
```

Chapter 10

Exercise 1

```
//: c10:E01_Rhomboid.java
/****************** Exercise 1 ******************
 * Add Rhomboid to Shapes.java. Create a
 * Rhomboid, upcast it to a Shape, then downcast
 * it back to a Rhomboid. Try downcasting to a
 * Circle and see what happens.
 ***********************************************/
import java.util.*;

class Shape {
  void draw() {
    System.out.println(this + ".draw()");
  }
}

class Circle extends Shape {
  public String toString() { return "Circle"; }
}

class Square extends Shape {
  public String toString() { return "Square"; }
}

class Triangle extends Shape {
  public String toString() { return "Triangle"; }
}

class Rhomboid extends Shape {
  public String toString() { return "Rhomboid"; }
}

public class E01_Rhomboid {
  public static void main(String[] args) {
    List s = new ArrayList();
    s.add(new Circle());
    s.add(new Square());
    s.add(new Triangle());
    s.add(new Rhomboid());
    Iterator e = s.iterator();
    while(e.hasNext())
      ((Shape)e.next()).draw();
    // Upcast to shape:
    Shape shape = new Rhomboid();
    // Downcast to Rhomboid:
    Rhomboid r = (Rhomboid)shape;
    // Downcast to Circle. Succeeds at compile time,
    // but fails at runtime with a ClassCastException:
    //! Circle c = (Circle)shape;
  }
} ///:~
```

The output is:

```
Circle.draw()
Square.draw()
Triangle.draw()
Rhomboid.draw()
```

Exercise 2

```
//: c10:E02_Instanceof.java
/****************** Exercise 2 ******************
 * Modify Exercise 1 so that it uses instanceof
 * to check the type before performing the
 * downcast.
 ***********************************************/
public class E02_Instanceof {
  public static void main(String[] args) {
    // Upcast to shape:
    Shape shape = new Rhomboid();
    // Downcast to Rhomboid:
    Rhomboid r = (Rhomboid)shape;
    // Test before Downcasting:
    Circle c = null;
    if(shape instanceof Circle)
      c = (Circle)shape;
    else
      System.out.println("shape not a Circle");
  }
} ///:~
```

The output is:

```
shape not a Circle
```

Exercise 3

```
//: c10:E03_Highlight.java
/****************** Exercise 3 ******************
 * Modify Shapes.java so that it can "highlight"
 * (set a flag) in all shapes of a particular
 * type. The toString() method for each derived
 * Shape should indicate whether that Shape is
 * "highlighted."
 ***********************************************/
import java.util.*;
import java.lang.reflect.*;

class HShape {
  boolean highlighted = false;
  public void highlight() { highlighted = true; }
  public void clearHighlight() { highlighted = false; }
  public void draw() {
    System.out.println(this + " draw()");
  }
  public String toString() {
    return getClass().getName() +
      (highlighted ? " highlighted " : " normal");
  }
  static ArrayList allShapes = new ArrayList();
  public HShape() { allShapes.add(this); }
  // Basic approach (code duplication)
  public static void highlight1(Class type) {
    Iterator it = allShapes.iterator();
    while(it.hasNext()) {
      HShape shape = (HShape)it.next();
      if(type.isInstance(shape))
        shape.highlight();
    }
  }
  public static void clearHighlight1(Class type) {
    Iterator it = allShapes.iterator();
    while(it.hasNext()) {
      HShape shape = (HShape)it.next();
      if(type.isInstance(shape))
        shape.clearHighlight();
    }
  }
  // Create an applicator and reuse it. All exceptions
  // indicate programmer error, and are thus
  // RuntimeExceptions:
  public static void forEach(Class type, String method) {
    try {
      Method m = HShape.class.getMethod(method, null);
      Iterator it = allShapes.iterator();
      while(it.hasNext()) {
        HShape shape = (HShape)it.next();
        if(type.isInstance(shape))
          m.invoke(shape, null);
      }
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public static void highlight2(Class type) {
    forEach(type, "highlight");
  }
  public static void clearHighlight2(Class type) {
    forEach(type, "clearHighlight");
  }
}

class HCircle extends HShape {}
class HSquare extends HShape {}
class HTriangle extends HShape {}
```

```
public class E03_Highlight {
  public static void main(String[] args) {
    List s = new ArrayList(
      Arrays.asList(new HShape[]{
        new HCircle(), new HSquare(),
        new HTriangle(), new HSquare(),
        new HTriangle(), new HCircle(),
        new HCircle(), new HSquare(), }));
    HShape.highlight1(HCircle.class);
    HShape.highlight2(HCircle.class);
    Iterator e = s.iterator();
    while(e.hasNext())
      ((HShape)e.next()).draw();
    System.out.println("*******");
    // Highlight them all:
    HShape.highlight1(HShape.class);
    HShape.highlight2(HShape.class);
    e = s.iterator();
    while(e.hasNext())
      ((HShape)e.next()).draw();
    System.out.println("*******");
    // Not in the hierarchy:
    HShape.clearHighlight1(ArrayList.class);
    HShape.clearHighlight2(ArrayList.class);
    e = s.iterator();
    while(e.hasNext())
      ((HShape)e.next()).draw();
  }
} ///:~
```

The first thing I've done in this example is to move all the methods into the base class, because it's possible and it makes sense to eliminate any code that appears to be duplicated. This consisted of moving **toString( )** into the base class, and using RTTI to determine the name of the class. Also, **toString( )** prints information (as specified) about whether the object is "highlighted" or not.

Highlighting is determined on a per-object basis with a **boolean** flag, which can be set or cleared using methods in **HShape**. However, to be able to ask all objects of a particular type to be highlighted or cleared, you must keep track of all objects of this class. This is done with a **static**

ArrayList called **allShapes**, and an **HShape** default constructor that adds the current object to **allShapes**.

The **static** method **highlight1( )** takes an argument of type **Class** which is the type to highlight. It iterates through **allShapes** and uses **Class.isInstance( )** to see if the argument matches with each object; if so it calls **highlight( )** for that object. The **clearHighlight1( )** method works the same way.

Note the redundant code in **highlight1( )** and **clearHighlight1( )**. As an alternative approach, I've used reflection in **forEach( )** to combine the redundant code used in iterating through the list and applying the operation to each element in the list. **Highlight2( )** and **clearHighlight2( )** each call **forEach( )**.

In **main( )**, I've taken a different approach to populating the **ArrayList** of **HShape** objects: I created an array, then used **java.util.Arrays.asList( )** to turn it into a list, then passed that to the **ArrayList** constructor that takes a **Collection** and copies it into the new **ArrayList**. This eliminates typing when you're populating large lists.

To call **HShape.highlight( )**, you pass the name of the specific **HShape** you want to highlight, with **.class** appended to produce the **Class** reference. Notice that if you pass **HShape.class** as the argument, it matches every **HShape** in the list, so all are highlighted. You can use **clearHighlight( )** in this way to clear all highlighting.

If you pass a class that isn't in the **HShape** hierarchy, the **highlight** and **clearHighlight** methods quietly do nothing, since the classes never match.

Exercise 4

```
//: c10:E04_CommandLoad.java
// {Args: c10.Gum c10.Cookie}
/****************** Exercise 4 ******************
 * Modify SweetShop.java so that each type of
 * object creation is controlled by a
 * command-line argument. That is, if your
 * command line is "java SweetShop Candy," then
 * only the Candy object is created. Notice how
 * you can control which Class objects are loaded
 * via the command-line argument.
 ***********************************************/
package c10;

class Candy {
  static {
    System.out.println("Loading Candy");
  }
}

class Gum {
  static {
    System.out.println("Loading Gum");
  }
}

class Cookie {
  static {
    System.out.println("Loading Cookie");
  }
}

public class E04_CommandLoad {
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < args.length; i++)
      Class.forName(args[i]);
  }
} ///:~
```

The output for the command line:

```
java E04_CommandLoad c10.Gum c10.Cookie
```

is:

```
Loading Gum
Loading Cookie
```

Exercise 5

```
//: c10:E05_NewPet.java
/****************** Exercise 5 ******************
 * Add a new type of Pet to PetCount3.java.
 * Verify that it is created and counted
 * correctly in main().
 ***********************************************/
package c10;
import java.util.*;

class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

// New type of pet:
class Labrador extends Dog {} // Added this line

class Counter { int i; }

public class E05_NewPet {
  public static void main(String[] args) {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      Pet.class,
      Dog.class,
      Pug.class,
      Labrador.class, // Had to add this line too
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw new RuntimeException(e);
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw new RuntimeException(e);
    }
    HashMap h = new HashMap();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      // Using isInstance to eliminate individual
      // instanceof expressions:
      for(int j = 0; j < petTypes.length; ++j)
        if(petTypes[j].isInstance(o)) {
          String key = petTypes[j].toString();
          ((Counter)h.get(key)).i++;
        }
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
      String nm = (String)keys.next();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

I just copied the code from the book example, and added the two lines shown above (can you figure out a way to add only one line – the new class?).

The output from one run is:

```
class Pug
class Labrador
class Cat
class Pug
class Dog
class Labrador
class Dog
class Gerbil
class Cat
class Dog
class Rodent
class Pug
class Dog
class Pug
class Labrador
class Labrador quantity: 3
class Hamster quantity: 0
class Gerbil quantity: 1
class Cat quantity: 2
class Pet quantity: 15
class Pug quantity: 4
class Rodent quantity: 2
class Dog quantity: 11
```

Exercise 6

```
//: c10:E06_RecursiveClassPrint.java
// {Args: Hamster FancyToy}
/****************** Exercise 6 ******************
 * Write a method that takes an object and
 * recursively prints all the classes in that
 * object's hierarchy.
 ***********************************************/
public class E06_RecursiveClassPrint {
  static void printClasses(Class c) {
    // getSuperclass() returns null on Object:
    if(c == null) return;
    System.out.println(c.getName());
    // Produces the interfaces that this class
    // implements:
    Class[] interfaces = c.getInterfaces();
    for(int i = 0; i < interfaces.length; i++)  {
      Class k = interfaces[i];
      System.out.println("Interface: " + k.getName());
      printClasses(k.getSuperclass());
    }
    printClasses(c.getSuperclass());
  }
  public static void main(String args[]) throws Exception {
    for(int i = 0; i < args.length; i++) {
      System.out.println("Displaying " + args[i]);
      printClasses(Class.forName("c10." + args[i]));
      if(i < args.length -1)
        System.out.println("==================");
    }
  }
} ///:~
```

I've enhanced the solution a bit from the specification by also printing out all the interfaces. To demonstrate the printing of interfaces, I've run the program on **FancyToy**, which is part of Exercise 8.

The output of the program for the command line:

```
java E06_RecursiveClassPrint Hamster FancyToy
```

is:

```
Displaying Hamster
Hamster
Rodent
Pet
java.lang.Object
==================
Displaying FancyToy
FancyToy
Interface: HasBatteries
```

```
Interface: Waterproof
Interface: ShootsThings
Toy
java.lang.Object
```

Exercise 7

```
//: c10:E07_GetDeclaredFields.java
// {Args: c10.Derived}
/****************** Exercise 7 ******************
 * Modify Exercise 6 so that it uses
 * Class.getDeclaredFields() to also display
 * information about the fields in a class.
 ***********************************************/
package c10;
import java.lang.reflect.*;
import java.util.*;

interface Iface {
  int i = 47;
  void f();
}

class Base implements Iface {
  String s;
  double d;
  public void f() {
    System.out.println("Base.f");
  }
}

class Composed {
  Base b;
}

class Derived extends Base {
  Composed c;
  String s;
}
```

```
public class E07_GetDeclaredFields {
  static Set alreadyDisplayed = new HashSet();
  static void printClasses(Class c) {
    // getSuperclass() returns null on Object:
    if(c == null) return;
    System.out.println(c.getName());
    Field[] fields = c.getDeclaredFields();
    if(fields.length != 0)
      System.out.println("Fields:");
    for(int i = 0; i < fields.length; i++) {
      System.out.println("\t" + fields[i]);
    }
    for(int i = 0; i < fields.length; i++) {
      Class field = fields[i].getType();
      if(!alreadyDisplayed.contains(field)) {
        printClasses(field);
        alreadyDisplayed.add(field);
      }
    }
    // Produces the interfaces that this class
    // implements:
    Class[] interfaces = c.getInterfaces();
    for(int i = 0; i < interfaces.length; i++)  {
      Class k = interfaces[i];
      System.out.println("Interface: " + k.getName());
      printClasses(k.getSuperclass());
    }
    printClasses(c.getSuperclass());
  }
  public static void main(String args[])
  throws Exception {
    for(int i = 0; i < args.length; i++) {
      System.out.println("Displaying " + args[i]);
      printClasses(Class.forName(args[i]));
      if(i < args.length -1)
        System.out.println("==================");
    }
  }
} ///:~
```

I began by creating an interesting class hierarchy which consists of interfaces and composition of complex types.

Initially I added the code beginning with the line containing **getDeclaredFields( )**, but then I found that the output got a bit tedious so I added a **Set** for **Field**s that had been displayed already (so they aren't displayed multiple times).

Exercise 8

```
//: c10:E08_ToyDefault.java
/****************** Exercise 8 ******************
 * In ToyTest.java, comment out Toy's default
 * constructor and explain what happens.
 ***********************************************/
package c10;

interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}

class Toy {
  // Comment out the following default
  // constructor to see
  // NoSuchMethodError from (*1*)
  Toy() {} // Comment this line to get the effect
  Toy(int i) {}
}

class FancyToy extends Toy
    implements HasBatteries,
      Waterproof, ShootsThings {
  FancyToy() { super(1); }
}

public class E08_ToyDefault {
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("c10.FancyToy");
    } catch(ClassNotFoundException e) {
      System.err.println("Can't find FancyToy");
      throw new RuntimeException(e);
    }
    printInfo(c);
    Class[] faces = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Requires default constructor:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw new RuntimeException(e);
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw new RuntimeException(e);
    }
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println("Class name: " + cc.getName() +
      " is interface? [" + cc.isInterface() + "]");
  }
} ///:~
```

In its original form (with the default constructor intact) the output is:

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
With the default constructor commented out, the output is:
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Cannot instantiate
```

```
Exception in thread "main"
java.lang.InstantiationException: Toy
        at java.lang.Class.newInstance0(Native Method)
        at java.lang.Class.newInstance(Unknown Source)
        at E08_ToyDefault.main(E08_ToyDefault.java:42)
```

As the code says, **cy.newInstance( )** requires a default constructor, and since there exists a non-default constructor the compiler won't synthesize one for you if you don't explicitly define one.

Exercise 9

```
//: c10:E09_NewToyInterface.java
/****************** Exercise 9 ******************
 * Incorporate a new kind of interface into
 * ToyTest.java and verify that it is detected
 * and displayed properly.
 ***********************************************/
package c10;

interface HasCPU {}

class FancierToy extends Toy implements HasBatteries,
  HasCPU, Waterproof, ShootsThings {
  FancierToy() { super(1); }
}

public class E09_NewToyInterface {
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("c10.FancierToy");
    } catch(ClassNotFoundException e) {
      System.err.println("Can't find FancierToy");
      throw new RuntimeException(e);
    }
    printInfo(c);
    Class[] faces = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Requires default constructor:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw new RuntimeException(e);
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw new RuntimeException(e);
    }
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println("Class name: " + cc.getName() +
      " is interface? [" + cc.isInterface() + "]");
  }
} ///:~
```

The output is:

```
Class name: FancierToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: HasCPU is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
```

Exercise 10

```
//: c10:E10_PrimitiveOrTrue.java
/****************** Exercise 10 *****************
 * Write a program to determine whether an array
 * of char is a primitive type or a true object.
 ***********************************************/
public class E10_PrimitiveOrTrue {
  public static void main(String args[]) {
    char[] ac = "Hello, World!".toCharArray();
    System.out.println("ac.getClass() = " + ac.getClass());
    System.out.println("ac.getClass().getSuperclass() = "
      + ac.getClass().getSuperclass());
    char c = 'c';
    //! c.getClass(); // Can't do it, primitives
                      // are not true objects.
    int[] ia = new int[3];
    System.out.println("ia.getClass() = " + ia.getClass());
    long[] la = new long[3];
    System.out.println("la.getClass() = " + la.getClass());
    double[] da = new double[3];
    System.out.println("da.getClass() = " + da.getClass());
    String[] sa = new String[3];
    System.out.println("sa.getClass() = " + sa.getClass());
    E10_PrimitiveOrTrue[] pot =
      new E10_PrimitiveOrTrue[3];
    System.out.println(
      "pot.getClass() = " + pot.getClass());
    // Multi-dimensional arrays:
    int[][][] threed = new int[3][][];
    System.out.println(
      "threed.getClass() = " + threed.getClass());
  }
} ///:~
```

I created the array of **char** by defining a string and calling **toCharArray( )** on it. Note that I can call **getClass( )** and **getSuperclass( )** for **ac** because it is an object of a true class, but if I try to call **getClass( )** for a primitive like **char c**, the compiler complains. Thus, not all elements of Java are objects, so when you hear claims that Java is a "pure" OO language, remember this example.

The reason I continued on to look at other arrays is to see what kinds of output you get when you print the class names for these arrays. The output from the program is:

```
ac.getClass() = class [C
ac.getClass().getSuperclass() = class java.lang.Object
ia.getClass() = class [I
la.getClass() = class [J
da.getClass() = class [D
sa.getClass() = class [Ljava.lang.String;
pot.getClass() = class [LE11_PrimitiveOrTrue;
threed.getClass() = class [[[I
```

It appears that all array class names begin with '**[**'. This is followed by a one-character code for primitive types, and an **L** followed by the type of element contained in the array for arrays of objects.

For multi-dimensional arrays, there's a '**[**' for each dimension. You can learn more about low-level details like this in *Inside the Java 2 Virtual Machine*, by Bill Venners (McGraw-Hill 1999).

Exercise 11

```
//: c10:E11_ClearSpitValve.java
/****************** Exercise 11 *****************
 * Implement clearSpitValve() as described in the
 * summary.
 ***********************************************/
// The summary reads:
/*
One option is to put a clearSpitValve() method
in the base class Instrument, but this is
confusing because it implies that Percussion and
Electronic instruments also have spit valves.
RTTI provides a much more reasonable solution in
this case because you can place the method in the
specific class (Wind in this case), where it's
appropriate. However, a more appropriate solution
is to put a prepareInstrument() method in the
base class, but you might not see this when
you're first solving the problem and could
mistakenly assume that you must use RTTI.
*/
// We'll use the last-defined version of the
// instrument hierarchy:
interface Instrument {
  int i = 5; // static & final
  void play();
  String what();
  void adjust();
  void prepareInstrument();
}
```

```
class Wind implements Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
  public void clearSpitValve() {
    System.out.println("Wind.clearSpitValve");
  }
  public void prepareInstrument() {
    clearSpitValve();
  }
}

class Percussion implements Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
  public void prepareInstrument() {
    System.out.println("Percussion.prepareInstrument");
  }
}

class Stringed implements Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
  public void prepareInstrument() {
    System.out.println("Stringed.prepareInstrument");
  }
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
  public void clearSpitValve() {
    System.out.println("Brass.clearSpitValve");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
  public void clearSpitValve() {
    System.out.println("Woodwind.clearSpitValve");
  }
}

class Music5 {
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  static void prepareAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      e[i].prepareInstrument();
  }
}

public class E11_ClearSpitValve {
  public static void main(String[] args) {
    Instrument[] orchestra = {
      new Wind(), new Percussion(),
      new Stringed(), new Brass(),
      new Woodwind(),
    };
    Music5.prepareAll(orchestra);
    Music5.tuneAll(orchestra);
  }
} ///:~
```

The output is:

```
Wind.clearSpitValve
Percussion.prepareInstrument
Stringed.prepareInstrument
Brass.clearSpitValve
Woodwind.clearSpitValve
Wind.play()
Percussion.play()
Stringed.play()
Brass.play()
Woodwind.play()
```

Exercise 12

```
//: c10:E12_RotateShape.java
/****************** Exercise 12 *****************
 * Implement the rotate(Shape) method described
 * in this chapter, such that it checks to see if
 * it is rotating a Circle (and, if so, doesn't
 * perform the operation).
 ***********************************************/
// This exercise contained an error -- the
// rotate() method was not in fact described
// in the chapter.
import java.util.*;

class RShape {
  void draw() {
    System.out.println(this + ".draw()");
  }
  void rotate(int degrees) {
    System.out.println("Rotating " + this +
      " by " + degrees + " degrees");
  }
  public String toString() {
    return getClass().getName();
  }
}

class RCircle extends RShape {}

class RSquare extends RShape {}

class RTriangle extends RShape {}

class RRhomboid extends RShape {}

public class E12_RotateShape {
  public static void rotateAll(
    Iterator it, int degrees) {
    while(it.hasNext()) {
      RShape s = (RShape)it.next();
      if(!(s instanceof RCircle))
        s.rotate(degrees);
    }
  }
  public static void main(String args[]) {
    List s = new ArrayList(
      Arrays.asList(new RShape[]{
        new RCircle(), new RRhomboid(),
        new RTriangle(), new RSquare(),
        new RRhomboid(), new RSquare(),
        new RTriangle(), new RCircle(),
        new RCircle(), new RSquare(),
      }));
    rotateAll(s.iterator(), 45);
  }
} ///:~
```

The output is:

```
Rotating RRhomboid by 45 degrees
Rotating RTriangle by 45 degrees
Rotating RSquare by 45 degrees
Rotating RRhomboid by 45 degrees
Rotating RSquare by 45 degrees
Rotating RTriangle by 45 degrees
Rotating RSquare by 45 degrees
```

Notice that no circles are being rotated.

Exercise 13

```
//: c10:E13_ReflectionToyCreation.java
/****************** Exercise 13 *****************
 * In ToyTest.java, use reflection to create a
 * Toy object using the nondefault constructor.
 ***********************************************/
package c10;
import java.lang.reflect.*;

class SuperToy extends FancierToy {
  int IQ;
  public SuperToy(int intelligence) { IQ = intelligence; }
  public String toString() {
    return "I'm a SuperToy. I'm smarter than you";
  }
}

public class E13_ReflectionToyCreation {
  public static Toy
  makeToy(String toyName, int IQ) {
    try {
      Class tClass = Class.forName("c10." + toyName);
      Constructor[] ctors = tClass.getConstructors();
      for(int i = 0; i < ctors.length; i++) {
        // Look for a constructor with a single parameter:
        Class[] params = ctors[i].getParameterTypes();
        if(params.length == 1)
          if(params[0] == int.class)
            return (Toy)ctors[i].newInstance(
              new Object[]{ new Integer(IQ)} );
      }
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    return null;
  }
  public static void main(String args[]) {
    System.out.println(makeToy("SuperToy", 150));
  }
} ///:~
```

This is one approach, albeit a rather limited one. If you want to get fancier, once you get the types of the arguments for the constructor you can ask the user (via the console) what values should be used.

The output is:

```
I'm a SuperToy. I'm smarter than you
```

Exercise 14

```
//: c10:E14_ClassDump.java
// {Args: java.lang.String c10.SuperToy}
/****************** Exercise 14 *****************
 * Look up the interface for java.lang.Class in
 * the HTML Java documentation from java.sun.com.
 * Write a program that takes the name of a class
 * as a command-line argument, then uses the
 * Class methods to dump all the information
 * available for that class. Test your program
 * with a standard library class and a class you
 * create.
 ***********************************************/
package c10;
// The solution is a much-modified version of
// Showmethods.java.

public class E14_ClassDump {
  public static void display(String msg, Object[] vals) {
    System.out.println(msg);
    for(int i = 0; i < vals.length; i++)
      System.out.println("    " + vals[i]);
  }
  public static void classInfo(Class c) throws Exception {
    System.out.println("c.getName(): " + c.getName());
    System.out.println("c.getPackage(): " +
      c.getPackage());
    System.out.println("c.getSuperclass(): " +
      c.getSuperclass());
    // This produces all the classes declared as members:
    display("c.getDeclaredClasses()",
      c.getDeclaredClasses());
    display("c.getClasses()", c.getClasses());
    display("c.getInterfaces()", c.getInterfaces());
    // The "Declared" version includes all
    // versions, not just public:
    display("c.getDeclaredMethods()",
      c.getDeclaredMethods());
    display("c.getMethods()", c.getMethods());
    display("c.getDeclaredConstructors()",
      c.getDeclaredConstructors());
    display("c.getConstructors()", c.getConstructors());
    display("c.getDeclaredFields()",
      c.getDeclaredFields());
    display("c.getFields()", c.getFields());
    if(c.getSuperclass() != null) {
      System.out.println("Base class: --------");
      classInfo(c.getSuperclass());
    }
    System.out.println("End of " + c.getName());
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < args.length; i++)
      classInfo(Class.forName(args[i]));
  }
} ///:~
```

This is a reasonable sample of the methods in **Class** that give reflection information, although there are a few more you could add. Also, I've made it recursively go up the inheritance hierarchy. The output for **SuperToy** is:

```
c.getName(): SuperToy
c.getPackage(): null
c.getSuperclass(): class FancierToy
c.getDeclaredClasses()
c.getClasses()
c.getInterfaces()
c.getDeclaredMethods()
```

```
   public java.lang.String SuperToy.toString()
c.getMethods()
   public native int java.lang.Object.hashCode()
   public final void java.lang.Object.wait() throws
java.lang.InterruptedException
   public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
   public final native void java.lang.Object.wait(long)
throws java.lang.InterruptedException
   public final native java.lang.Class
java.lang.Object.getClass()
   public boolean java.lang.Object.equals(java.lang.Object)
   public final native void java.lang.Object.notify()
   public final native void java.lang.Object.notifyAll()
   public java.lang.String SuperToy.toString()
c.getDeclaredConstructors()
   public SuperToy(int)
c.getConstructors()
   public SuperToy(int)
c.getDeclaredFields()
   int SuperToy.IQ
c.getFields()
Base class: --------
c.getName(): FancierToy
c.getPackage(): null
c.getSuperclass(): class Toy
c.getDeclaredClasses()
c.getClasses()
c.getInterfaces()
   interface HasBatteries
   interface HasCPU
   interface Waterproof
   interface ShootsThings
c.getDeclaredMethods()
c.getMethods()
   public native int java.lang.Object.hashCode()
   public final void java.lang.Object.wait() throws
java.lang.InterruptedException
   public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
```

```
   public final native void java.lang.Object.wait(long)
throws java.lang.InterruptedException
   public final native java.lang.Class
java.lang.Object.getClass()
   public boolean java.lang.Object.equals(java.lang.Object)
   public java.lang.String java.lang.Object.toString()
   public final native void java.lang.Object.notify()
   public final native void java.lang.Object.notifyAll()
c.getDeclaredConstructors()
   FancierToy()
c.getConstructors()
c.getDeclaredFields()
c.getFields()
Base class: --------
c.getName(): Toy
c.getPackage(): null
c.getSuperclass(): class java.lang.Object
c.getDeclaredClasses()
c.getClasses()
c.getInterfaces()
c.getDeclaredMethods()
c.getMethods()
   public native int java.lang.Object.hashCode()
   public final void java.lang.Object.wait() throws
java.lang.InterruptedException
   public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
   public final native void java.lang.Object.wait(long)
throws java.lang.InterruptedException
   public final native java.lang.Class
java.lang.Object.getClass()
   public boolean java.lang.Object.equals(java.lang.Object)
   public java.lang.String java.lang.Object.toString()
   public final native void java.lang.Object.notify()
   public final native void java.lang.Object.notifyAll()
c.getDeclaredConstructors()
   Toy(int)
   Toy()
c.getConstructors()
c.getDeclaredFields()
c.getFields()
```

```
Base class: --------
c.getName(): java.lang.Object
c.getPackage(): package java.lang, Java Platform API
Specification, version 1.3
c.getSuperclass(): null
c.getDeclaredClasses()
c.getClasses()
c.getInterfaces()
c.getDeclaredMethods()
   public native int java.lang.Object.hashCode()
   protected void java.lang.Object.finalize() throws
java.lang.Throwable
   public final void java.lang.Object.wait() throws
java.lang.InterruptedException
   public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
   public final native void java.lang.Object.wait(long)
throws java.lang.InterruptedException
   private static native void
java.lang.Object.registerNatives()
   public final native java.lang.Class
java.lang.Object.getClass()
   public boolean java.lang.Object.equals(java.lang.Object)
   protected native java.lang.Object
java.lang.Object.clone() throws
java.lang.CloneNotSupportedException
   public java.lang.String java.lang.Object.toString()
   public final native void java.lang.Object.notify()
   public final native void java.lang.Object.notifyAll()
c.getMethods()
   public native int java.lang.Object.hashCode()
   public final void java.lang.Object.wait() throws
java.lang.InterruptedException
   public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
   public final native void java.lang.Object.wait(long)
throws java.lang.InterruptedException
   public final native java.lang.Class
java.lang.Object.getClass()
   public boolean java.lang.Object.equals(java.lang.Object)
   public java.lang.String java.lang.Object.toString()
   public final native void java.lang.Object.notify()
   public final native void java.lang.Object.notifyAll()
c.getDeclaredConstructors()
   public java.lang.Object()
c.getConstructors()
   public java.lang.Object()
c.getDeclaredFields()
c.getFields()
End of java.lang.Object
End of Toy
End of FancierToy
End of SuperToy
```

Try it with something more complex, like **javax.swing.JCheckBox**.

Exercise 15

```
//: c10:E15_ShowMethods2.java
// {Args: c10.E15_ShowMethods2}
/********************* Exercise 15 *********************
 * Modify the regular expression in ShowMethods.java to
 * additionally strip off the keywords native and final
 * (hint: Use the "or" operator '|')
 ********************************************************/
package c10;
import java.lang.reflect.*;
import java.util.regex.*;

public class E15_ShowMethods2 {
  private static final String usage =
    "usage: \n" +
    "ShowMethods pified.class.name\n" +
    "To show all methods in class or: \n" +
    "ShowMethods pified.class.name word\n" +
    "To search for methods involving 'word'";
  private static Pattern p =
    Pattern.compile("\\w+\\.|native\\s|final\\s");
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    int lines = 0;
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      if(args.length == 1) {
        for(int i = 0; i < m.length; i++)
          System.out.println(
            p.matcher(m[i].toString()).replaceAll(""));
        for(int i = 0; i < ctor.length; i++)
          System.out.println(
            p.matcher(ctor[i].toString()).replaceAll(""));
        lines = m.length + ctor.length;
      } else {
        for(int i = 0; i < m.length; i++)
          if(m[i].toString().indexOf(args[1]) != -1) {
            System.out.println(
              p.matcher(m[i].toString()).replaceAll(""));
            lines++;
          }
        for(int i = 0; i < ctor.length; i++)
          if(ctor[i].toString().indexOf(args[1]) != -1) {
            System.out.println(p.matcher(
              ctor[i].toString()).replaceAll(""));
            lines++;
          }
      }
    } catch(ClassNotFoundException e) {
      System.out.println("No such class: " + e);
    }
  }
} ///:~
```

Chapter 11

Exercise 1

```
//: c11:E01_RandDouble.java
/****************** Exercise 1 ******************
 * Create an array of double and fill() it using
 * RandDoubleGenerator. Print the results.
 ***********************************************/
import com.bruceeckel.util.Arrays2;

public class E01_RandDouble {
  public static void main(String args[]) {
    double[] da = new double[10];
    Arrays2.fill(da,
      new Arrays2.RandDoubleGenerator());
    System.out.println(Arrays2.toString(da));
  }
} ///:~
```

This is simply an exercise in using the generator from the package.

The output for one run is:

```
(0.5001203986291445, 0.8539982462192328,
0.6726664848706867, 0.892650339160878,
0.6006173265811632, 0.6156735136224198,
0.6478510992181377, 0.5858583665142717,
0.8760917695678225, 0.5931881088554638)
```

Exercise 2

```
//: c11:E02_Gerbil.java
/****************** Exercise 2 ******************
 * Create a new class called Gerbil with an int
 * gerbilNumber that's initialized in the
 * constructor (similar to the Mouse example in
 * this chapter). Give it a method called hop()
 * that prints out which gerbil number this is,
 * and that it's hopping. Create an ArrayList and
 * add a bunch of Gerbil objects to the List. Now
 * use the get() method to move through the List
 * and call hop() for each Gerbil.
 ***********************************************/
import java.util.*;

class Gerbil2 {
  private static int gerbilCounter = 0;
  private int gerbilNumber = ++gerbilCounter;
  public String toString() {
    return "gerbil " + gerbilNumber;
  }
  public void hop() {
    System.out.println(toString()
      + " is hopping");
  }
}

public class E02_Gerbil {
  public static void main(String args[]) {
    ArrayList gerbils = new ArrayList();
    for(int i = 0; i < 10; i++)
      gerbils.add(new Gerbil2());
    for(int i = 0; i < gerbils.size(); i++)
      ((Gerbil2)gerbils.get(i)).hop();
  }
} ///:~
```

The output is:

```
gerbil 1 is hopping
gerbil 2 is hopping
gerbil 3 is hopping
gerbil 4 is hopping
gerbil 5 is hopping
gerbil 6 is hopping
gerbil 7 is hopping
gerbil 8 is hopping
gerbil 9 is hopping
gerbil 10 is hopping
```

Exercise 3

```
//: c11:E03_GerbilIterator.java
/****************** Exercise 3 ******************
 * Modify Exercise 2 so you use an Iterator to
 * move through the List while calling hop().
 ***********************************************/
import java.util.*;

public class E03_GerbilIterator {
  public static void main(String args[]) {
    ArrayList gerbils = new ArrayList();
    for(int i = 0; i < 10; i++)
      gerbils.add(new Gerbil());
    for(Iterator it = gerbils.iterator();
        it.hasNext();)
      ((Gerbil)it.next()).hop();
  }
} ///:~
```

This produces the same output as before.

Exercise 4

```
//: c11:E04_GerbilMap.java
/****************** Exercise 4 ******************
 * Take the Gerbil class in Exercise 2 and put it
 * into a Map instead, associating the name of
 * the Gerbil as a String (the key) for each
 * Gerbil (the value) you put in the table. Get
 * an Iterator for the keySet() and use it to
 * move through the Map, looking up the Gerbil
 * for each key and printing out the key and
 * telling the gerbil to hop().
 ***********************************************/
import java.util.*;
```

```
public class E04_GerbilMap {
  public static void main(String args[]) {
    HashMap map = new HashMap();
    map.put("Bob", new Gerbil());
    map.put("Frank", new Gerbil());
    map.put("Tiffany", new Gerbil());
    map.put("Ted", new Gerbil());
    map.put("Wallace", new Gerbil());
    map.put("Heather", new Gerbil());
    Iterator it = map.keySet().iterator();
    while(it.hasNext()) {
      String key = (String)it.next();
      Gerbil value = (Gerbil)map.get(key);
      System.out.println("name = " + key
        + ", value = " + value);
    }
  }
} ///:~
```

The output is:

```
name = Frank, value = gerbil 2
name = Bob, value = gerbil 1
name = Wallace, value = gerbil 5
name = Ted, value = gerbil 4
name = Tiffany, value = gerbil 3
name = Heather, value = gerbil 6
```

Note that, because of the nature of the hashing function (described in the book), the objects are not stored in the order that they were entered. If you use a **TreeMap** instead, you'll see that the order *is* maintained.

Exercise 5

```
//: c11:E05_CountryList.java
/****************** Exercise 5 ******************
 * Create a List (try both ArrayList and
 * LinkedList) and fill it using
 * Collections2.countries. Sort the list and
 * print it, then apply Collections.shuffle() to
 * the list repeatedly, printing it each time so
 * that you can see how the shuffle() method
 * randomizes the list differently each time.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E05_CountryList {
  public static void main(String args[]) {
    List lst = new ArrayList();
    Collections2.fill(lst, Collections2.countries, 8);
    Collections.sort(lst);
    System.out.println(lst);
    for(int i = 0; i < 5; i++) {
      Collections.shuffle(lst);
      System.out.println(lst);
    }
  }
} ///:~
```

I only did it for **ArrayList**. You can easily change it (in only one place) for **LinkedList**.

Here's the output from one run:

```
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
CAMEROON, CAPE VERDE]
[BENIN, ANGOLA, ALGERIA, CAPE VERDE, BURUNDI, CAMEROON,
BOTSWANA, BURKINA FASO]
[ALGERIA, BURKINA FASO, CAPE VERDE, BURUNDI, BOTSWANA,
BENIN, ANGOLA, CAMEROON]
[BENIN, ALGERIA, CAPE VERDE, CAMEROON, ANGOLA, BOTSWANA,
BURUNDI, BURKINA FASO]
[BOTSWANA, BURKINA FASO, ALGERIA, CAMEROON, CAPE VERDE,
BURUNDI, ANGOLA, BENIN]
[ALGERIA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI, CAPE
VERDE, CAMEROON, ANGOLA]
```

Exercise 6

```
//: c11:E06_MouseListRestriction.java
/****************** Exercise 6 ******************
 * Demonstrate that you can't add anything but a
 * Mouse to a MouseList.
 ***********************************************/
import java.util.*;

class MouseList {
  private ArrayList list = new ArrayList();
  public void add(Mouse2 m) {
    list.add(m);
  }
  public Mouse2 get(int index) {
    return (Mouse2)list.get(index);
  }
  public int size() { return list.size(); }
}

public class E06_MouseListRestriction {
  public static void main(String args[]) {
    MouseList ml = new MouseList();
    // Compile-time error: not a Mouse2
    //! ml.add(new Object());
  }
} ///:~
```

Here is **Mouse2.java**

```
//: c11:Mouse2.java

public class Mouse2 {
  private int mouseNumber;
  Mouse2(int i) { mouseNumber = i; }
  // Override Object.toString():
  public String toString() {
    return "This is Mouse #" + mouseNumber;
  }
  public int getNumber() {
    return mouseNumber;
  }
} ///:~
```

Exercise 7

```
//: c11:E07_MouseListProblem.java
/****************** Exercise 7 ******************
 * Modify MouseList.java so that it inherits from
 * ArrayList instead of using composition.
 * Demonstrate the problem with this approach.
 ***********************************************/
import java.util.*;

class MouseList2 extends ArrayList {
  public void add(Mouse2 m) {
    System.out.println("Adding a mouse");
    super.add(m);
  }
  // Can't override with different return type:
  //! public Mouse2 get(int i) {
  //!  return (Mouse2)super.get(i);
  //! }
  // Sidestep:
  public Mouse2 getMouse(int i) {
    return (Mouse2)super.get(i);
  }
}

public class E07_MouseListProblem {
  public static void main(String args[]) {
    MouseList2 mice = new MouseList2();
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse2(i));
    for(int i = 0; i < mice.size(); i++)
      System.out.println(mice.getMouse(i));
    // Oops! Can add a non-mouse:
    mice.add(new Object());
  }
} ///:~
```

The compiler detects a problem when trying to override **get( )** because the only thing that's different is the return value. We can "cleverly" sidestep this by making a **getMouse( )** method. However, note that the **ArrayList.get( )** method is still alive and well, so it's still possible to pull objects out through that channel.

Here's the output:

```
Adding a mouse
Adding a mouse
Adding a mouse
This is Mouse #0
This is Mouse #1
This is Mouse #2
```

Note that the **Object** does not cause an "Adding..." message to be printed, because the **add( )** method is actually *overloading* rather than *overriding*, which means that the **add(Object)** method still exists, and is called in the last case. Inheriting from **ArrayList** thus does not limit the type of objects that can be added, and so it definitely isn't what we want. And with a non-**Mouse** in the list, when you call **getMouse( )** with that object an exception will be thrown during the cast.

Exercise 8

```
//: c11:E08_RepairCatsAndDogs.java
/****************** Exercise 8 ******************
 * Repair CatsAndDogs.java by creating a Cats
 * container (utilizing ArrayList) that will only
 * accept and retrieve Cat objects.
 ***********************************************/
import java.util.*;

class Cat {
  private int catNumber;
  Cat(int i) { catNumber = i; }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
}
```

```
class Dog {
  private int dogNumber;
  Dog(int i) { dogNumber = i; }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
}

class CatContainer {
  private ArrayList cats = new ArrayList();
  public void add(Cat c) { cats.add(c); }
  public Cat get(int i) {
    return (Cat)cats.get(i);
  }
  public int size() { return cats.size(); }
}

public class E08_RepairCatsAndDogs {
  public static void main(String[] args) {
    CatContainer cats = new CatContainer();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Will not accept a Dog:
    //! cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      cats.get(i).print();
    // Dog is detected at compile-time
  }
} ///:~
```

Exercise 9

```
//: c11:E09_MapOrder.java
/****************** Exercise 9 ************************
 * Fill a HashMap with key-value pairs. Print the results
 * to show ordering by hash code. Extract the pairs, sort
 * by key, and place the result into a LinkedHashMap. Show
 * that the insertion order is maintained.
 *******************************************************/
```

```
import com.bruceeckel.util.*;
import java.util.*;

public class E09_MapOrder {
  public static void main(String[] args) {
    Map m1 = new HashMap();
    Collections2.fill(m1, Collections2.geography, 25);
    System.out.println(m1);
    Object keys[] = m1.keySet().toArray();
    Arrays.sort(keys);
    Map m2 = new LinkedHashMap();
    for(int i = 0; i < keys.length; i++)
      m2.put(keys[i], m1.get(keys[i]));
    System.out.println(m2);
  }
} ///:~
```

Exercise 10

```
//: c11:E10_SetOrder.java
/****************** Exercise 10 *****************
 * Repeat the previous example with a HashSet and
 * LinkedHashSet.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E10_SetOrder {
  public static void main(String[] args) {
    Set s1 = new HashSet();
    Collections2.fill(s1, Collections2.capitals, 25);
    System.out.println(s1);
    Object elements[] = s1.toArray();
    Arrays.sort(elements);
    Set s2 = new LinkedHashSet();
    for(int i = 0; i < elements.length; i++)
      s2.add(elements[i]);
    System.out.println(s2);
  }
} ///:~
```

Exercise 11

```
//: c11:E11_TypedContainer.java
/****************** Exercise 11 *****************
 * Create a new type of container that uses a
 * private ArrayList to hold the objects. Using a class
 * reference, Capture the type of the first object you put
 * in it, and then allow the user to insert objects of
 * only that type from then on.
 ***********************************************/
import java.util.*;

class TypedContainer extends AbstractList {
  private ArrayList list = new ArrayList();
  Class type = null;
  public boolean add(Object o) {
    if(type == null)
      type = o.getClass();
    if(type.isAssignableFrom(o.getClass())) {
      list.add(o);
      return true;
    }
    return false;
  }
  public void add(int index, Object o) {
    if(type == null)
      type = o.getClass();
    if(type.isAssignableFrom(o.getClass()))
      list.add(index, o);
  }
  public Object set(int index, Object o) {
    if(type == null)
      type = o.getClass();
    if(type.isAssignableFrom(o.getClass()))
      return list.set(index, o);
    return list.get(index);
  }
  public Object get(int index) {
    return list.get(index);
  }
  public int size() { return list.size(); }
  public Object remove(int index) {
    return list.remove(index);
  }
  public int indexOf(Object o) {
    return list.indexOf(o);
  }
  public int lastIndexOf(Object o) {
    return list.lastIndexOf(o);
  }
  public void clear() {
    list.clear();
  }
  public boolean addAll(int index, Collection c) {
    return list.addAll(index, c);
  }
  public Iterator iterator() {
    return list.iterator();
  }
  public ListIterator listIterator() {
    return list.listIterator();
  }
  public ListIterator listIterator(int index) {
    return list.listIterator(index);
  }
  public List subList(int fromIndex, int toIndex) {
    return list.subList(fromIndex, toIndex);
  }
  public boolean equals(Object o) {
    return list.equals(o);
  }
}

class X {
  public String toString() {
    return "X";
  }
}

class Y {
  public String toString() {
    return "Y";
  }
}

class Z extends X {
  public String toString() {
    return "Z";
  }
}

public class E11_TypedContainer {
  public static void main(String args[]) {
    TypedContainer tc = new TypedContainer();
    for(int i = 0; i < 10; i++) {
      tc.add(new X());
      tc.add(new Y());
    }
    System.out.println("tc = " + tc);
    TypedContainer tc2 = new TypedContainer();
    for(int i = 0; i < 10; i++) {
      tc2.add(new Y());
      tc2.add(new X());
    }
    System.out.println("tc2 = " + tc2);
    TypedContainer tc3 = new TypedContainer();
    for(int i = 0; i < 10; i++) {
      tc3.add(new X());
      tc3.add(new Z()); // Allows adding subclass of X
    }
    System.out.println("tc3 = " + tc3);
    TypedContainer tc4 = new TypedContainer();
    for(int i = 0; i < 10; i++) {
      tc4.add(new Z());
      tc4.add(new X()); // Type X not allowed
    }
    System.out.println("tc4 = " + tc4);
  }
} ///:~
```

Exercise 12

```
//: c11:E12_StringContainer.java
import java.util.*;

/****************** Exercise 12 ******************
 * Create a container that encapsulates an array
 * of String, and that only adds Strings and gets
 * Strings, so that there are no casting issues
 * during use. If the internal array isn't big
 * enough for the next add, your container should
 * automatically resize it. In main(), compare
 * the performance of your container with an
 * ArrayList holding Strings.
 ***********************************************/
class StringContainer {
  private String[] array;
  private int index = 0;
  private static final int INCR = 255;
  public StringContainer() {
    array = new String[10];
  }
  public StringContainer(int sz) {
    array = new String[sz];
  }
  public void add(String s) {
    if(index >= array.length) {
      System.out.println("Resizing");
      String[] tmp =
        new String[array.length + INCR];
      for(int i = 0; i < array.length; i++)
        tmp[i] = array[i];
      index = array.length;
      array = tmp;
    }
    array[index++] = s;
  }
  public String get(int i) {
    if(i < index && i >= 0)
      return array[i];
    else
      return "Bad Index";
    // Use exceptions instead
  }
  public int size() {
    return index;
  }
}

public class E12_StringContainer {
  public static void main(String args[]) {
    String[] words = { "All", "good", "boys",
      "deserve", "fudge", "but",
      "not", "too", "much"
    };
    StringContainer sc = new StringContainer(5);
    for(int i = 0; i < words.length; i++)
      sc.add(words[i]);
    for(int i = 0; i < sc.size(); i++)
      System.out.print(sc.get(i) + " ");
    long t1 = System.currentTimeMillis();
    StringContainer sc2 =
      new StringContainer(100000);
    for(int i = 0; i < 100000; i++) {
      sc2.add(new Integer(i).toString());
      sc2.get(i);
    }
    long t2 = System.currentTimeMillis();
    System.out.println("StringContainer: "
      + (t2 - t1));
    t1 = System.currentTimeMillis();
    ArrayList lst = new ArrayList(100000);
    for(int i = 0; i < 100000; i++) {
      lst.add(new Integer(i).toString());
      lst.get(i);
    }
    t2 = System.currentTimeMillis();
    System.out.println("ArrayList: "
      + (t2 - t1));
  }
} ///:~
```

The test is very simple (and not terribly accurate), lifted from the book. The output for one run is:

```
Resizing
All good boys deserve fudge but not too much
StringContainer: 600
ArrayList: 660
```

You shouldn't take this too seriously, except to observe that the **ArrayList** performance is probably very close to anything you can hand-code, and so you shouldn't write containers yourself – use the ones in the standard library.

Exercise 13

```
//: c11:E13_IntContainer.java
/****************** Exercise 13 *****************
 * Repeat Exercise 12 for a container of int, and
 * compare the performance to an ArrayList
 * holding Integer objects. In your performance
 * comparison, include the process of
 * incrementing each object in the container.
 ***********************************************/
import java.util.*;

class IntContainer {
  private int[] array;
  private int index = 0;
  private static final int INCR = 255;
  public IntContainer() {
    array = new int[13];
  }
  public IntContainer(int sz) {
    array = new int[sz];
  }
  public void add(int s) {
    if(index >= array.length) {
      System.out.println("Resizing");
      int[] tmp =
        new int[array.length + INCR];
      for(int i = 0; i < array.length; i++)
        tmp[i] = array[i];
      index = array.length;
      array = tmp;
    }
    array[index++] = s;
  }
  public int get(int i) {
    if(i < index && i >= 0)
      return array[i];
    else
      throw new RuntimeException("Bad Index");
  }
  public void set(int i, int val) {
    if(i < index && i >= 0)
      array[i] = val;
    else
      throw new RuntimeException("Bad Index");
  }
  public int size() {
    return index;
  }
}

public class E13_IntContainer {
  public static void main(String args[]) {
    IntContainer ic = new IntContainer(5);
    for(int i = 0; i < 10; i++)
      ic.add(i);
    for(int i = 0; i < ic.size(); i++)
      System.out.print(ic.get(i) + " ");
    long t1 = System.currentTimeMillis();
    IntContainer ic2 =
      new IntContainer(100000);
    for(int i = 0; i < 100000; i++) {
      ic2.add(i);
      ic2.set(i, ic2.get(i) + 1);
    }
    long t2 = System.currentTimeMillis();
    System.out.println("IntContainer: "
      + (t2 - t1));
    t1 = System.currentTimeMillis();
    ArrayList lst = new ArrayList(100000);
    for(int i = 0; i < 100000; i++) {
      lst.add(new Integer(i));
      lst.set(i, new Integer(
        ((Integer)lst.get(i)).intValue() + 1));
    }
    t2 = System.currentTimeMillis();
    System.out.println("ArrayList: "
      + (t2 - t1));
  }
} ///:~
```

The output for one run is:

```
Resizing
0 1 2 3 4 5 6 7 8 9 IntContainer: 60
ArrayList: 220
```

The **IntContainer** appears to be significantly faster than the **ArrayList**.

Exercise 14

```
//: c11:E14_PrimitiveArrays.java
/****************** Exercise 14 *****************
 * Using the utilities in com.bruceeckel.util,
 * create an array of each primitive type and of
 * String, then fill each array using an
 * appropriate generator, and print each array
 * using the appropriate toString() method.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E14_PrimitiveArrays {
  public static void main(String[] args) {
    int size = 6;
    // Or get the size from the command line:
    if(args.length != 0)
      size = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays2.fill(a1,
      new Arrays2.RandBooleanGenerator());
    System.out.println("a1 = " + Arrays2.toString(a1));
    Arrays2.fill(a2,
      new Arrays2.RandByteGenerator());
    System.out.println("a2 = " + Arrays2.toString(a2));
    Arrays2.fill(a3,
      new Arrays2.RandCharGenerator());
    System.out.println("a3 = " + Arrays2.toString(a3));
    Arrays2.fill(a4,
      new Arrays2.RandShortGenerator());
    System.out.println("a4 = " + Arrays2.toString(a4));
    Arrays2.fill(a5,
      new Arrays2.RandIntGenerator());
    System.out.println("a5 = " + Arrays2.toString(a5));
    Arrays2.fill(a6,
      new Arrays2.RandLongGenerator());
    System.out.println("a6 = " + Arrays2.toString(a6));
    Arrays2.fill(a7,
      new Arrays2.RandFloatGenerator());
    System.out.println("a7 = " + Arrays2.toString(a7));
    Arrays2.fill(a8,
      new Arrays2.RandDoubleGenerator());
    System.out.println("a8 = " + Arrays2.toString(a8));
    Arrays2.fill(a9,
      new Arrays2.RandStringGenerator(7));
    System.out.println("a9 = " + Arrays.asList(a9));
  }
} ///:~
```

This is just a slightly edited version of **TestArrays2.java** from the book.

The output from one run is:

```
a1 =  (true, false, false, false, false, true)
a2 =  (40, -76, -35, -2, -70, -67)
a3 =  (k, R, M, W, h, b)
a4 =  (16456, -31639, 29351, -7199, -1347, 10638)
a5 =  (-3132, 6098, -2084, -9452, -6246, 3620)
a6 =  (-975983279247461340, -5334486128033659473,
-8993306774351743502, 6159205296425354234,
 7965276974414571894, 5826857740334035663)
a7 =  (0.626064, 0.9212824, 0.24075747,
0.46483123, 0.14160186, 0.07450378)
a8 =  (0.22488236931881733, 0.411608966866836,
0.4875233021888832, 0.44968495638586126,
0.09365339796588479, 0.01893378481136887)
a9 =  (NMSLXVC, FnpBWSG, uohbPqn, hJUdBPM,
ZvNshWa, llGNsuH)
```

Exercise 15

```
//: c11:E15_MovieNameGenerator.java
/****************** Exercise 15 *****************
 * Create a generator that produces character
 * names from your favorite movies (you can use
 * Snow White or Star Wars as a fallback), and
 * loops around to the beginning when it runs out
 * of names. Use the utilities in
 * com.bruceeckel.util to fill an array, an
 * ArrayList, a LinkedList, and both types of Set,
 * then print each container.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

class MovieNameGenerator implements Generator {
  String[] characters = {
    "Grumpy", "Happy", "Sleepy", "Dopey", "Doc", "Sneezy",
    "Bashful", "Snow White", "Witch Queen", "Prince"
  };
  int next = 0;
  public Object next() {
    String r = characters[next];
    next = (next + 1) % characters.length;
    return r;
  }
}

public class E15_MovieNameGenerator {
  public static void main(String args[]) {
    MovieNameGenerator mng = new MovieNameGenerator();

    String[] sa = new String[5];
    Arrays2.fill(sa, mng);
    System.out.println("sa = " + Arrays.asList(sa));

    ArrayList al = new ArrayList();
    Collections2.fill(al, mng, 5);
    System.out.println("al = " + al);

    LinkedList ll = new LinkedList();
    Collections2.fill(ll, mng, 5);
    System.out.println("ll = " + ll);

    HashSet hs = new HashSet();
    Collections2.fill(hs, mng, 5);
    System.out.println("hs = " + hs);

    TreeSet ts = new TreeSet();
    Collections2.fill(ts, mng, 5);
    System.out.println("ts = " + ts);
  }
} ///:~
```

The output is:

```
sa =  (Grumpy, Happy, Sleepy, Dopey, Doc)
al = [Sneezy, Bashful, Snow White,
Witch Queen, Prince]
ll = [Grumpy, Happy, Sleepy, Dopey, Doc]
hs = [Witch Queen, Prince, Sneezy,
Snow White, Bashful]
ts = [Doc, Dopey, Grumpy, Happy, Sleepy]
```

Exercise 16

```
//: c11:E16_ComparableClass.java
/****************** Exercise 16 *****************
 * Create a class containing two String objects,
 * and make it Comparable so that the comparison
 * only cares about the first String. Fill an
 * array and an ArrayList with objects of your
 * class, using the geography generator.
 * Demonstrate that sorting works properly. Now
 * make a Comparator that only cares about the
 * second String and demonstrate that sorting
 * works properly; also perform a binary search
 * using your Comparator.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

class TwoString implements Comparable {
  String s1, s2;
  public TwoString(String s1i, String s2i) {
    s1 = s1i;
    s2 = s2i;
  }
  public String toString() {
    return "[s1 = " + s1 + ", s2 = " + s2 + "]";
  }
  public int compareTo(Object rv) {
    String rvi = ((TwoString)rv).s1;
    return s1.compareTo(rvi);
  }
  private static Arrays2.RandStringGenerator gen =
    new Arrays2.RandStringGenerator(7);
  public static Generator generator() {
    return new Generator() {
      public Object next() {
        return new TwoString(
          (String)gen.next(),(String)gen.next());
      }
    };
  }
}

class CompareSecond implements Comparator {
  public int compare(Object o1, Object o2) {
    TwoString sc1 = (TwoString)o1;
    TwoString sc2 = (TwoString)o2;
    return sc1.s2.compareTo(sc2.s2);
  }
}

public class E16_ComparableClass {
  public static void main(String[] args) {
    TwoString[] array = new TwoString[10];
    Arrays2.fill(array, TwoString.generator());
    System.out.println("before sorting, array = " +
      Arrays.asList(array));
    Arrays.sort(array);
    System.out.println("\nafter sorting, array = " +
      Arrays.asList(array));
    Arrays.sort(array, new CompareSecond());
    System.out.println(
      "\nafter sorting with CompareSecond, array = " +
      Arrays.asList(array));

    ArrayList list = new ArrayList();
    Collections2.fill(list, TwoString.generator(), 10);
    TwoString zeroth = (TwoString)list.get(0);
    System.out.println("\nbefore sorting, list = " + list);
    Collections.sort(list);
    System.out.println("\nafter sorting, list = " + list);
    Comparator comp = new CompareSecond();
    Collections.sort(list, comp);
    System.out.println(
      "\nafter sorting with CompareSecond, list = "
      + list);

    int index =
      Collections.binarySearch(list, zeroth, comp);
    System.out.println("\nFormer zeroth element " +
      zeroth + " now located at " + index);
  }
} ///:~
```

This is a modification of the book example **CompType.java**. The generator is created using **RandStringGenerator**, and the **compareTo( )** is implemented using **String**'s **compareTo( )**.

The output from one run is:

```
before sorting, array = [[s1 = SEUvPjA, s2 = khsVtiQ], [s1
= WhZGMcr, s2 = gFwsYsz], [s1 = EDiYKSz, s2 = LbGIGUi], [s1
= FXBkhQm, s2 = FuHwcWW], [s1 = xdDWybn, s2 = ZqGZRTD], [s1
= IxkcuTm, s2 = wAYxddh], [s1 = gLenuQm, s2 = lzzCBJv], [s1
= SRgiodS, s2 = injlfJl], [s1 = vOpfnDP, s2 = RxKOuEJ], [s1
= VkBiLGR, s2 = joAEAIt]]

after sorting, array = [[s1 = EDiYKSz, s2 = LbGIGUi], [s1 =
FXBkhQm, s2 = FuHwcWW], [s1 = IxkcuTm, s2 = wAYxddh], [s1 =
SEUvPjA, s2 = khsVtiQ], [s1 = SRgiodS, s2 = injlfJl], [s1 =
VkBiLGR, s2 = joAEAIt], [s1 = WhZGMcr, s2 = gFwsYsz], [s1 =
gLenuQm, s2 = lzzCBJv], [s1 = vOpfnDP, s2 = RxKOuEJ], [s1 =
xdDWybn, s2 = ZqGZRTD]]

after sorting with CompareSecond, array = [[s1 = FXBkhQm,
s2 = FuHwcWW], [s1 = EDiYKSz, s2 = LbGIGUi], [s1 = vOpfnDP,
s2 = RxKOuEJ], [s1 = xdDWybn, s2 = ZqGZRTD], [s1 = WhZGMcr,
s2 = gFwsYsz], [s1 = SRgiodS, s2 = injlfJl], [s1 = VkBiLGR,
s2 = joAEAIt], [s1 = SEUvPjA, s2 = khsVtiQ], [s1 = gLenuQm,
s2 = lzzCBJv], [s1 = IxkcuTm, s2 = wAYxddh]]

before sorting, list = [[s1 = BsGEKVT, s2 = dIWruUe], [s1 =
pWNTCLV, s2 = lfdvCcn], [s1 = wHFpTtg, s2 = vvFiGgd], [s1 =
HNddKMM, s2 = byEFAKu], [s1 = qjrzMLb, s2 = FCUrIbz], [s1 =
GraECuq, s2 = OVqobtl], [s1 = NhQlRxy, s2 = fqOAGXq], [s1 =
HSAeYAz, s2 = oPSBimF], [s1 = DYaRZHS, s2 = NYWFqID], [s1 =
bHxrXlX, s2 = uflOLxf]]

after sorting, list = [[s1 = BsGEKVT, s2 = dIWruUe], [s1 =
DYaRZHS, s2 = NYWFqID], [s1 = GraECuq, s2 = OVqobtl], [s1 =
HNddKMM, s2 = byEFAKu], [s1 = HSAeYAz, s2 = oPSBimF], [s1 =
NhQlRxy, s2 = fqOAGXq], [s1 = bHxrXlX, s2 = uflOLxf], [s1 =
pWNTCLV, s2 = lfdvCcn], [s1 = qjrzMLb, s2 = FCUrIbz], [s1 =
wHFpTtg, s2 = vvFiGgd]]

after sorting with CompareSecond, list = [[s1 = qjrzMLb, s2
= FCUrIbz], [s1 = DYaRZHS, s2 = NYWFqID], [s1 = GraECuq, s2
= OVqobtl], [s1 = HNddKMM, s2 = byEFAKu], [s1 = BsGEKVT, s2
= dIWruUe], [s1 = NhQlRxy, s2 = fqOAGXq], [s1 = pWNTCLV, s2
= lfdvCcn], [s1 = HSAeYAz, s2 = oPSBimF], [s1 = bHxrXlX, s2
= uflOLxf], [s1 = wHFpTtg, s2 = vvFiGgd]]

Former zeroth element [s1 = BsGEKVT, s2 = dIWruUe] now
located at 4
```

Exercise 17

```
//: c11:E17_AlphaComparable.java
/****************** Exercise 17 *****************
 * Modify Exercise 16 so that an alphabetic sort
 * is used.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

class TwoStringAlphabetic implements Comparable {
  String s1, s2;
  public TwoStringAlphabetic(String s1i, String s2i) {
    s1 = s1i;
    s2 = s2i;
  }
  public String toString() {
    return "[s1 = " + s1 + ", s2 = " + s2 + "]";
  }
  public int compareTo(Object rv) {
    String rvi = ((TwoStringAlphabetic)rv).s1;
    return s1.toLowerCase().compareTo(rvi.toLowerCase());
  }
  private static Arrays2.RandStringGenerator gen =
    new Arrays2.RandStringGenerator(7);
  public static Generator generator() {
    return new Generator() {
      public Object next() {
        return new TwoStringAlphabetic(
          (String)gen.next(),(String)gen.next());
      }
    };
  }
}

class CompareSecondAlphabetic
implements Comparator {
  public int compare(Object o1, Object o2) {
    TwoStringAlphabetic sc1 = (TwoStringAlphabetic)o1;
    TwoStringAlphabetic sc2 = (TwoStringAlphabetic)o2;
    return sc1.s1.toLowerCase().compareTo(
      sc2.s1.toLowerCase());
  }
}

public class E17_AlphaComparable {
  public static void main(String[] args) {
    TwoStringAlphabetic[] array =
      new TwoStringAlphabetic[10];
    Arrays2.fill(array, TwoStringAlphabetic.generator());
    System.out.println("before sorting, array = " +
      Arrays.asList(array));
    Arrays.sort(array);
    System.out.println("\nafter sorting, array = " +
      Arrays.asList(array));
    Arrays.sort(array, new CompareSecondAlphabetic());
    System.out.println(
      "\nafter sorting with CompareSecondAlphabetic, " +
      "array = " + Arrays.asList(array));

    ArrayList list = new ArrayList();
    Collections2.fill(list,
      TwoStringAlphabetic.generator(), 10);
    TwoStringAlphabetic zeroth =
      (TwoStringAlphabetic)list.get(0);
    System.out.println("\nbefore sorting, list = " + list);
    Collections.sort(list);
    System.out.println("\nafter sorting, list = " + list);
    Comparator comp = new CompareSecondAlphabetic();
    Collections.sort(list, comp);
    System.out.println(
      "\nafter sorting with CompareSecondAlphabetic, " +
      "list = " + list);
    int index =
      Collections.binarySearch(list, zeroth, comp);
    System.out.println("\nFormer zeroth element " +
      zeroth + " now located at " + index);
  }
} ///:~
```

This uses a modification of the book's **AlphabeticComparator**. Note that **s1** and **s2** in **TwoStringAlphabetic** must have package access in order for the **CompareSecondAlphabetic** to access the values, otherwise accessors would be necessary.

The output from one run is:

```
before sorting, array = [[s1 = xiFYKDA, s2 = YftIuQO], [s1
= qWHiyPK, s2 = IkzdJ
lq], [s1 = vEBdjQq, s2 = rOBCFwl], [s1 = pwVASnn, s2 =
kRFywoY], [s1 = bzzQycx,
s2 = rbMoxbr], [s1 = ZpJeSqb, s2 = XEAnsJb], [s1 = NhDnLDn,
s2 = WleazkL], [s1 =
 CsphMHE, s2 = PGXoust], [s1 = cklJHbY, s2 = afWAiLS], [s1
= epAmdis, s2 = JJEbo
JP]]

after sorting, array = [[s1 = bzzQycx, s2 = rbMoxbr], [s1 =
cklJHbY, s2 = afWAiL
S], [s1 = CsphMHE, s2 = PGXoust], [s1 = epAmdis, s2 =
JJEboJP], [s1 = NhDnLDn, s
2 = WleazkL], [s1 = pwVASnn, s2 = kRFywoY], [s1 = qWHiyPK,
s2 = IkzdJlq], [s1 =
vEBdjQq, s2 = rOBCFwl], [s1 = xiFYKDA, s2 = YftIuQO], [s1 =
ZpJeSqb, s2 = XEAnsJ
b]]

after sorting with CompareSecondAlphabetic, array = [[s1 =
bzzQycx, s2 = rbMoxbr
], [s1 = cklJHbY, s2 = afWAiLS], [s1 = CsphMHE, s2 =
PGXoust], [s1 = epAmdis, s2
 = JJEboJP], [s1 = NhDnLDn, s2 = WleazkL], [s1 = pwVASnn,
s2 = kRFywoY], [s1 = q
WHiyPK, s2 = IkzdJlq], [s1 = vEBdjQq, s2 = rOBCFwl], [s1 =
xiFYKDA, s2 = YftIuQO
], [s1 = ZpJeSqb, s2 = XEAnsJb]]

before sorting, list = [[s1 = JZOlUOZ, s2 = gDDQGFu], [s1 =
zAhWKhL, s2 = oMGjKy
M], [s1 = pBIepJd, s2 = VBsiLSN], [s1 = suIRzuf, s2 =
nyKjgkJ], [s1 = ymwuuCi, s
2 = BWxDxOw], [s1 = HSXkKUc, s2 = FUugUCm], [s1 = lozejhn,
s2 = AqXcrYm], [s1 =
GtHsZZQ, s2 = JvCJIHf], [s1 = tGOTKmw, s2 = VMcHklc], [s1 =
VAFoEoY, s2 = qPtqLn
F]]

after sorting, list = [[s1 = GtHsZZQ, s2 = JvCJIHf], [s1 =
HSXkKUc, s2 = FUugUCm
], [s1 = JZOlUOZ, s2 = gDDQGFu], [s1 = lozejhn, s2 =
AqXcrYm], [s1 = pBIepJd, s2
 = VBsiLSN], [s1 = suIRzuf, s2 = nyKjgkJ], [s1 = tGOTKmw,
s2 = VMcHklc], [s1 = V
AFoEoY, s2 = qPtqLnF], [s1 = ymwuuCi, s2 = BWxDxOw], [s1 =
zAhWKhL, s2 = oMGjKyM
]]

after sorting with CompareSecondAlphabetic, list = [[s1 =
GtHsZZQ, s2 = JvCJIHf]
, [s1 = HSXkKUc, s2 = FUugUCm], [s1 = JZOlUOZ, s2 =
gDDQGFu], [s1 = lozejhn, s2
= AqXcrYm], [s1 = pBIepJd, s2 = VBsiLSN], [s1 = suIRzuf, s2
= nyKjgkJ], [s1 = tG
OTKmw, s2 = VMcHklc], [s1 = VAFoEoY, s2 = qPtqLnF], [s1 =
ymwuuCi, s2 = BWxDxOw]
, [s1 = zAhWKhL, s2 = oMGjKyM]]

Former zeroth element [s1 = JZOlUOZ, s2 = gDDQGFu] now
located at 2
```

Exercise 18

```
//: c11:E18_RandTreeSet.java
/****************** Exercise 18 *****************
 * Use Arrays2.RandStringGenerator to fill a
 * TreeSet but using alphabetic ordering. Print
 * the TreeSet to verify the sort order.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E18_RandTreeSet {
  static Arrays2.RandStringGenerator gen =
    new Arrays2.RandStringGenerator(7);
  public static void main(String args[]) {
    TreeSet ts = new TreeSet(new AlphabeticComparator());
    Collections2.fill(ts, gen, 10);
    System.out.println("ts = " + ts);
  }
} ///:~
```

The output from one run is:

```
ts = [bmIfXOe, FKEqeYF, gKrRSBJ, HFpxOoo,
hlFpviK, IhSzLnM, KvRMYab, rtWOkIu, tsCORAC,
xpVKSPF]
```

Exercise 19

```
//: c11:E19_CrossInsertion.java
/****************** Exercise 19 *****************
 * Create both an ArrayList and a LinkedList, and
 * fill each using the Collections2.capitals
 * generator. Print each list using an ordinary
 * Iterator, then insert one list into the other
 * using a ListIterator, inserting at every other
 * location. Now perform the insertion starting
 * at the end of the first list and moving
 * backward.
 ***********************************************/
```

```
import com.bruceeckel.util.*;
import java.util.*;

public class E19_CrossInsertion {
  public static void main(String args[]) {
    ArrayList al = new ArrayList();
    LinkedList ll = new LinkedList();
    Collections2.fill(al, Collections2.capitals, 10);
    Collections2.fill(ll, Collections2.capitals, 10);
    for(Iterator it=al.iterator(); it.hasNext();)
      System.out.println(it.next());
    System.out.println("********");
    for(Iterator it=ll.iterator(); it.hasNext();)
      System.out.println(it.next());
    System.out.println("********");
    int alindex = 0;
    for(ListIterator lit2 = ll.listIterator();
      lit2.hasNext();) {
      al.add(alindex, lit2.next());
      alindex += 2;
    }
    System.out.println("al = " + al);
    System.out.println("********");
    alindex = 0;
    // Start at the end:
    for(ListIterator lit2 =
      ll.listIterator(ll.size());
      lit2.hasPrevious();) {
      al.add(alindex, lit2.previous());
      alindex += 2;
    }
    System.out.println("al = " + al);
  }
} ///:~
```

This solution is a little risky as it is written for two lists of the same size (however, you'll get an exception if there is a mis-sizing, so it's sloppy but not terrible).

The output is:

```
Algiers
```

```
Luanda
Porto-Novo
Gaberone
Ouagadougou
Bujumbura
Yaounde
Praia
Bangui
N'djamena
********
Moroni
Brazzaville
Dijibouti
Cairo
Malabo
Asmara
Addis Ababa
Libreville
Banjul
Accra
********
al = [Moroni, Algiers, Brazzaville, Luanda,
Dijibouti, Porto-Novo, Cairo, Gaberone, Malabo,
Ouagadougou, Asmara, Bujumbura, Addis Ababa,
Yaounde, Libreville, Praia, Banjul, Bangui,
Accra, N'djamena]
********
al = [Accra, Moroni, Banjul, Algiers, Libreville,
Brazzaville, Addis Ababa, Luanda, Asmara,
Dijibouti, Malabo, Porto-Novo, Cairo, Cairo, Dijibouti,
Gaberone, Brazzaville, Malabo, Moroni,
Ouagadougou, Asmara, Bujumbura, Addis Ababa,
Yaounde, Libreville, Praia, Banjul, Bangui,
Accra, N'djamena]
```

Exercise 20

```
//: c11:E20_IterHashcode.java
/****************** Exercise 20 *****************
 * Write a method that uses an Iterator to step
 * through a Collection and print the hashCode()
 * of each object in the container. Fill all the
 * different types of Collections with objects
 * and apply your method to each container.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

public class E20_IterHashcode {
  public static void PrintHashes(Iterator it) {
    while(it.hasNext())
      System.out.println(it.next().hashCode());
  }
  public static void main(String args[]) {
    Collection[] ca = {
      new ArrayList(),
      new LinkedList(),
      new HashSet(),
      new TreeSet(),
    };
    for(int i = 0; i < ca.length; i++)
      Collections2.fill(ca[i], Collections2.capitals, 10);
    for(int i = 0; i < ca.length; i++)
      PrintHashes(ca[i].iterator());
  }
} ///:~
```

Exercise 21

```
//: c11:E21_RepairInfinite.java
/****************** Exercise 21 *****************
 * Repair the problem in InfiniteRecursion.java.
 ***********************************************/
import java.util.*;

class NoRecursion {
  public String toString() {
    return " NoRecursion address: "
      + super.toString() + "\n";
  }
}

public class E21_RepairInfinite {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new NoRecursion());
    System.out.println(v);
  }
} ///:~
```

Exercise 22

```
//: c11:E22_TestList.java
/****************** Exercise 22 *****************
 * Create a class, then make an initialized array
 * of objects of your class. Fill a List from
 * your array. Create a subset of your List using
 * subList(), and then remove this subset from
 * your List using removeAll().
 ***********************************************/
import java.util.*;

class IDClass {
  private static int counter = 0;
  private int count = counter++;
  public String toString() {
    return "IDClass " + count;
  }
}

public class E22_TestList {
  public static void main(String args[]) {
    IDClass[] idc = new IDClass[10];
    for(int i = 0; i < idc.length; i++)
      idc[i] = new IDClass();
    List lst = new ArrayList(Arrays.asList(idc));
    System.out.println("lst = " + lst);
    List subSet = new ArrayList(
      lst.subList(lst.size()/4, lst.size()/2));
    System.out.println("subSet = " + subSet);
    lst.removeAll(subSet);
    System.out.println("lst = " + lst);
  }
} ///:~
```

The methods **asList( )** and **subList( )** both return **List**s, but these are immutable (because they are "backed by" the underlying array) and you'll get a concurrent modification exception if you try to use them as shown above. So in both cases, I passed the immutable **List** objects into **ArrayList** constructors, and the objects are then copied (automatically, by the constructor) into the new **ArrayList**, which can then be modified as needed.

The output is:

```
lst = [IDClass 0, IDClass 1, IDClass 2, IDClass 3,
IDClass 4, IDClass 5, IDClass 6, IDClass 7,
IDClass 8, IDClass 9]
subSet = [IDClass 2, IDClass 3, IDClass 4]
lst = [IDClass 0, IDClass 1, IDClass 5, IDClass 6,
IDClass 7, IDClass 8, IDClass 9]
```

Exercise 23

```
//: c11:E23_RodentIterator.java
/****************** Exercise 23 *****************
 * Change Exercise 6 in Chapter 7 to use an
 * ArrayList to hold the Rodents and an Iterator
 * to move through the sequence of Rodents.
 * Remember that an ArrayList holds only Objects
 * so you must use a cast when accessing
 * individual Rodents.
 ***********************************************/
import java.util.*;

class Mouse extends Rodent {
  public void hop() {
    System.out.println("Mouse hopping");
  }
  public void scurry() {
    System.out.println("Mouse scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Mice");
  }
  public String toString() {
    return "Mouse";
  }
}

class Hamster extends Rodent {
  public void hop() {
    System.out.println("Hamster hopping");
  }
  public void scurry() {
    System.out.println("Hamster scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Hamsters");
  }
  public String toString() {
    return "Hamster";
  }
}

public class E23_RodentIterator {
  public static void main(String args[]) {
    // Typical way to create & populate a List:
    // ArrayList lst = new ArrayList();
    // lst.add(new Mouse());
    // lst.add(new Gerbil());
    // lst.add(new Hamster());
    // Using a dynamic array for initialization:
    ArrayList lst = new ArrayList(
      Arrays.asList(new Rodent[] {
        new Mouse(), new Gerbil(), new Hamster()
      }));
    Iterator it = lst.iterator();
    while(it.hasNext()) {
      Rodent r = (Rodent)it.next();
      r.hop();
      r.scurry();
      r.reproduce();
      System.out.println("r = " + r);
    }
  }
} ///:~
```

Here is **Rodent.java**

```
//: c11:Rodent.java

public class Rodent {
  public void hop() {
    System.out.println("Rodent hopping");
  }
  public void scurry() {
    System.out.println("Rodent scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Rodents");
  }
  public String toString() {
    return "Rodent";
  }
} ///:~
```

Here is **Gerbil.java**

```
//: c11:Gerbil.java

public class Gerbil extends Rodent {
  public void hop() {
    System.out.println("Gerbil hopping");
  }
  public void scurry() {
    System.out.println("Gerbil scurrying");
  }
  public void reproduce() {
    System.out.println("Making more Gerbils");
  }
  public String toString() {
    return "Gerbil";
  }
} ///:~
```

In **main( )**, the commented-out portion shows the typical way you might initialize the list – create the list and then repeatedly call **add( )**. However, you can also use the more succinct second approach, where you create the array dynamically, pass it to **Arrays.asList( )** and then pass the result of that (which is immutable) to the **ArrayList** constructor. If you do this, adding another element simply means adding another item to your list. This can be a convenient trick for initializing a **Container** from a list of elements.

The output is:

```
Mouse hopping
Mouse scurrying
Making more Mice
r = Mouse
Gerbil hopping
Gerbil scurrying
Making more Gerbils
r = Gerbil
Hamster hopping
Hamster scurrying
Making more Hamsters
r = Hamster
```

Exercise 24

```
//: c11:E24_Deque.java
/****************** Exercise 24 *****************
 * Following the Queue.java example, create a
 * Deque class and test it.
 ***********************************************/
import java.util.*;

// A full implementation of Collection:
class Deque implements Collection {
  private LinkedList list;
  public Deque() {
    list = new LinkedList();
  }
  // Copy from an existing Collection:
  public Deque(Collection c) {
    list = new LinkedList(c);
  }
  public void put_front(Object v) {
    list.addFirst(v);
  }
  public void put_back(Object v) {
    list.addLast(v);
  }
  public Object get_front() {
    return list.removeFirst();
  }
  public Object get_back() {
    return list.removeLast();
  }
  public boolean isEmpty() {
    return list.isEmpty();
  }
  public int size() { return list.size(); }
  public boolean contains(Object o) {
    return list.contains(o);
  }
  public Iterator iterator() {
    return list.iterator();
  }
  public Object[] toArray() {
    return list.toArray();
  }
  public Object[] toArray(Object a[]) {
    return list.toArray(a);
  }
  public boolean add(Object o) {
    list.addLast(o);
    return true;
  }
  public boolean remove(Object o) {
    return list.remove(o);
  }
  public boolean containsAll(Collection c) {
    return list.containsAll(c);
  }
  public boolean addAll(Collection c) {
    return list.addAll(c);
  }
  public boolean removeAll(Collection c) {
    return list.removeAll(c);
  }
  public boolean retainAll(Collection c) {
    return list.retainAll(c);
  }
  public void clear() { list.clear(); }
}

public class E24_Deque {
  public static void main(String[] args) {
    Deque deque1 = new Deque();
    for(int i = 0; i < 5; i++)
      deque1.put_front(Integer.toString(i));
    for(int i = 50; i < 55; i++)
      deque1.put_back(Integer.toString(i));
    Deque deque2 = new Deque(deque1);
    while(!deque1.isEmpty())
      System.out.println(deque1.get_front());
    System.out.println("**********");
    while(!deque2.isEmpty())
      System.out.println(deque2.get_back());
  }
} ///:~
```

Here, I went ahead and fully implemented the **Collection** interface, primarily because I wanted to be able to duplicate **deque1** into **deque2** once I'd filled **deque1** as you can see in **main( )**.

The code you see in **Deque** is often referred to as *delegation*, because I'm creating an implementation of one interface (**Collection**) using composition (with the **LinkedList**), and it means I have to write out all the methods as you see above (many IDEs, such as www.Eclipse.org, will do this automatically). This is a situation where one must strongly resist the urge to simply inherit from **LinkedList**, for reasons shown earlier in this book. At the same time, note that I rather mindlessly implemented some of the **Collection** methods; upon further reflection you may actually want to implement these differently or even perform no-ops (I'm not yet comfortable throwing an **UnsupportedOperationException**, and yet that may be the proper solution in some cases).

The output is:

```
4
3
2
1
0
50
51
52
53
54
**********
54
53
52
51
50
0
1
2
3
4
```

Exercise 25

```
//: c11:E25_TreeStatistics.java
/****************** Exercise 25 *****************
 * Use a TreeMap in Statistics.java. Now add code
 * that tests the performance difference between
 * HashMap and TreeMap in that program.
 ***********************************************/
import java.util.*;
```

```
class Counter3 {
  int i = 1;
  public String toString() {
    return Integer.toString(i);
  }
}

public class E25_TreeStatistics {
  public static void test(String type, Map m) {
    long t1 = System.currentTimeMillis();
    for(int i = 0; i < 100000; i++) {
      // Produce a number between 0 and 20:
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(m.containsKey(r))
        ((Counter3)m.get(r)).i++;
      else
        m.put(r, new Counter3());
    }
    long t2 = System.currentTimeMillis();
    System.out.println(m);
    System.out.println(type + ": " + (t2 - t1));
  }
  public static void main(String[] args) {
    test("HashMap", new HashMap());
    test("TreeMap", new TreeMap());
  }
} ///:~
```

Here, I moved all the code into the **test( )** method, then added some simple timing and ran it on both types of **Map**. The output is:

```
{19=5062, 18=5056, 17=5065, 16=4997, 15=5057,
14=4952, 13=5031, 12=5036, 11=4945, 10=5039,
9=5050, 8=5050, 7=4909, 6=5003, 5=4973, 4=4911,
3=4965, 2=4914, 1=5058, 0=4927}
HashMap: 50
{0=4979, 1=5037, 2=5054, 3=4890, 4=5049, 5=4961,
6=4922, 7=5009, 8=5099, 9=5147, 10=4949,
11=5022, 12=4926, 13=5088, 14=4975, 15=4908,
16=4982, 17=4970, 18=5037, 19=4996}
TreeMap: 110
```

So by this test, **HashMap** is about twice as fast as **TreeMap**. Keep in mind that a simple test like this doesn't reflect the full breadth of performance differences, and especially remember that the performance difference will change from one application to another, depending on how a **Map** is used.

Exercise 26

```
//: c11:E26_ACountries.java
/****************** Exercise 26 *****************
 * Produce a Map and a Set containing all the
 * countries that begin with 'A.'
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E26_ACountries {
  public static void main(String args[]) {
    TreeMap map = new TreeMap();
    TreeSet set = new TreeSet();
    Collections2.fill(map,Collections2.geography,
      CountryCapitals.pairs.length);
    Collections2.fill(set,Collections2.countries,
      CountryCapitals.pairs.length);
    Iterator it = map.keySet().iterator();
    String b = null;
    while(it.hasNext()) {
      String s = (String)it.next();
      if(s.startsWith("B")) {
        b = s;
        break;
      }
    }
    Map aMap = map.headMap(b);
    Set aSet = set.headSet(b);
    System.out.println("aMap = " + aMap);
    System.out.println("aSet = " + aSet);
  }
} ///:~
```

The easiest way to approach this problem is to use the fact that the tree implementations of **Map** and **Set** are automatically sorted. Both are filled with the country information, then an iterator from one container (it doesn't matter which since they both sort themselves) is used to find the first string that doesn't begin with 'A'. This is used in the **TreeMap** method **headMap( )** and the **TreeSet** method **headSet( )** (which you can find described in the JDK HTML documentation) to create the subsets.

The output is:

```
aMap = {AFGHANISTAN=Kabul, ALBANIA=Tirana,
ALGERIA=Algiers, ANDORRA=Andorra la Vella,
ANGOLA=Luanda, ANTIGUA AND BARBUDA=Saint John's,
ARGENTINA=Buenos Aires, ARMENIA=Yerevan,
AUSTRALIA=Canberra, AUSTRIA=Vienna,
AZERBAIJAN=Baku}
aSet = [AFGHANISTAN, ALBANIA, ALGERIA, ANDORRA,
ANGOLA, ANTIGUA AND BARBUDA, ARGENTINA, ARMENIA,
AUSTRALIA, AUSTRIA, AZERBAIJAN]
```

Exercise 27

```
//: c11:E27_VerifySet.java
/****************** Exercise 27 *****************
 * Using Collections2.countries, fill a Set
 * multiple times with the same data and verify
 * that the Set ends up with only one of each
 * instance. Try this with both kinds of Set.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E27_VerifySet {
  public static void main(String args[]) {
    TreeSet set = new TreeSet();
    Collections2.fill(
      set, Collections2.countries.reset(), 10);
    Collections2.fill(
      set, Collections2.countries.reset(), 10);
    Collections2.fill(
      set, Collections2.countries.reset(), 10);
    System.out.println("set = " + set);
  }
} ///:~
```

Note that you need to call **reset( )** (which returns the generator object after resetting the index) each time to cause the duplicate data to be entered. The output is:

```
set = [ALGERIA, ANGOLA, BENIN, BOTSWANA,
BURKINA FASO, BURUNDI, CAMEROON, CAPE VERDE,
CENTRAL AFRICAN REPUBLIC, CHAD]
```

And by this you can verify that the set is working properly.

Exercise 28

```
//: c11:E28_MoreProbable.java
/****************** Exercise 28 *****************
 * Starting with Statistics.java, create a
 * program that runs the test repeatedly and
 * looks to see if any one number tends to appear
 * more than the others in the results.
 ***********************************************/
import java.util.*;

class Counter {
  int i = 1;
  public String toString() {
    return Integer.toString(i);
  }
}

class HistoUnit implements Comparable {
  Counter counter;
  Integer val;
  public HistoUnit(Counter counter, Integer val) {
    this.counter = counter;
    this.val = val;
  }
  public int compareTo(Object o) {
    int lv = ((HistoUnit)o).counter.i;
    int rv = counter.i;
    return (lv < rv ? -1 : (lv == rv ? 0 : 1));
  }
  public String toString() {
    return "Value = " + val
      + ", Occurrences = " + counter.i + "\n";
  }
}

public class E28_MoreProbable {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10000000; i++) {
      // Produce a number between 0 and 100:
      Integer r =
        new Integer((int)(Math.random() * 100));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
      else
        hm.put(r, new Counter());
    }
    // Make a histogram:
    List lst = new ArrayList();
    Iterator it = hm.keySet().iterator();
    while(it.hasNext()) {
      Integer key = (Integer)it.next();
      lst.add(new HistoUnit(
        (Counter)hm.get(key), key));
    }
    Collections.sort(lst);
    System.out.println("lst = " + lst);
  }
} ///:~
```

I wanted to use as much built-in functionality as I could, so I just made a list of objects and used **Collections.sort( )** on that list. The object is **HistoUnit**, and you can see that its sorting is based on the value of **Counter**. Once the sort is performed you can just print out the list to see the results ordered by occurrence.

Here's the result of one run:

```
lst = [Value = 41, Occurrences = 100815
, Value = 9, Occurrences = 100579
, Value = 43, Occurrences = 100500
, Value = 36, Occurrences = 100482
, Value = 16, Occurrences = 100449
, Value = 64, Occurrences = 100442
, Value = 39, Occurrences = 100400
, Value = 95, Occurrences = 100394
, Value = 94, Occurrences = 100363
, Value = 67, Occurrences = 100359
, Value = 82, Occurrences = 100354
, Value = 33, Occurrences = 100352
, Value = 55, Occurrences = 100334
, Value = 76, Occurrences = 100330
, Value = 65, Occurrences = 100309
, Value = 98, Occurrences = 100300
, Value = 27, Occurrences = 100299
, Value = 44, Occurrences = 100291
, Value = 77, Occurrences = 100269
, Value = 91, Occurrences = 100263
, Value = 4, Occurrences = 100255
, Value = 92, Occurrences = 100232
, Value = 50, Occurrences = 100217
, Value = 83, Occurrences = 100206
, Value = 20, Occurrences = 100203
, Value = 6, Occurrences = 100199
, Value = 11, Occurrences = 100198
, Value = 53, Occurrences = 100177
, Value = 90, Occurrences = 100173
, Value = 59, Occurrences = 100171
, Value = 63, Occurrences = 100169
, Value = 66, Occurrences = 100168
, Value = 0, Occurrences = 100165
, Value = 48, Occurrences = 100153
, Value = 3, Occurrences = 100150
, Value = 37, Occurrences = 100146
, Value = 18, Occurrences = 100134
, Value = 25, Occurrences = 100115
, Value = 13, Occurrences = 100115
, Value = 51, Occurrences = 100111
, Value = 56, Occurrences = 100108
, Value = 93, Occurrences = 100098
, Value = 74, Occurrences = 100088
, Value = 26, Occurrences = 100079
, Value = 5, Occurrences = 100064
, Value = 57, Occurrences = 100062
, Value = 46, Occurrences = 100062
, Value = 72, Occurrences = 100033
, Value = 10, Occurrences = 100033
, Value = 22, Occurrences = 100026
, Value = 42, Occurrences = 100025
, Value = 69, Occurrences = 100021
, Value = 23, Occurrences = 100018
, Value = 2, Occurrences = 100011
, Value = 15, Occurrences = 100007
, Value = 80, Occurrences = 99984
, Value = 31, Occurrences = 99984
, Value = 24, Occurrences = 99975
, Value = 45, Occurrences = 99973
, Value = 62, Occurrences = 99965
, Value = 8, Occurrences = 99962
, Value = 81, Occurrences = 99960
, Value = 29, Occurrences = 99959
, Value = 84, Occurrences = 99949
, Value = 7, Occurrences = 99924
, Value = 54, Occurrences = 99910
, Value = 1, Occurrences = 99892
, Value = 85, Occurrences = 99881
, Value = 61, Occurrences = 99851
, Value = 75, Occurrences = 99850
, Value = 88, Occurrences = 99842
, Value = 97, Occurrences = 99829
, Value = 49, Occurrences = 99826
, Value = 38, Occurrences = 99814
, Value = 68, Occurrences = 99789
, Value = 40, Occurrences = 99781
, Value = 73, Occurrences = 99773
, Value = 34, Occurrences = 99769
, Value = 70, Occurrences = 99754
, Value = 30, Occurrences = 99731
, Value = 58, Occurrences = 99728
, Value = 17, Occurrences = 99715
, Value = 79, Occurrences = 99696
, Value = 19, Occurrences = 99670
, Value = 32, Occurrences = 99663
, Value = 14, Occurrences = 99633
, Value = 28, Occurrences = 99618
, Value = 86, Occurrences = 99612
, Value = 87, Occurrences = 99607
, Value = 12, Occurrences = 99562
, Value = 99, Occurrences = 99555
, Value = 71, Occurrences = 99549
, Value = 78, Occurrences = 99541
, Value = 52, Occurrences = 99510
, Value = 60, Occurrences = 99508
, Value = 21, Occurrences = 99465
, Value = 89, Occurrences = 99438
, Value = 96, Occurrences = 99369
, Value = 35, Occurrences = 99319
, Value = 47, Occurrences = 99239
]
```

There does not appear to be any preferred value. If you're statistically inclined, you can calculate the distribution and standard deviation to determine how good the randomization function is.

Exercise 29

```java
//: c11:E29_HashStatistics.java
/****************** Exercise 29 *****************
 * Rewrite Statistics.java using a HashSet of
 * Counter objects (you'll have to modify Counter
 * so that it will work in the HashSet). Which
 * approach seems better?
 ***********************************************/
import java.util.*;

class Counter2 {
  int count = 1;
  Integer value;
  public Counter2(int val) {
    this.value = new Integer(val);
  }
  public String toString() {
    return "value: " + value + ", occurrences: "
      + Integer.toString(count) + " hashcode: "
      + hashCode() + "\n";
  }
  public int hashCode() { return value.hashCode(); }
  public boolean equals(Object obj) {
    return obj instanceof Counter2 &&
      ((Counter2)obj).value.equals(value);
      // This doesn't work!:
      //! ((Counter2)obj).value == value;
  }
}

public class E29_HashStatistics {
  public static void main(String[] args) {
    HashSet hs = new HashSet();
    for(int i = 0; i < 100000; i++) {
      // Produce a number between 0 and 20:
      Counter2 c =
        new Counter2((int)(Math.random() * 20));
      if(hs.contains(c)) {
        // The only way to access the set
        // elements is with an iterator:
        Iterator it = hs.iterator();
        while(it.hasNext()) {
          Counter2 ig = (Counter2)it.next();
          //! if(ig == c) { // This doesn't work!
          if(ig.equals(c)) {
            ig.count++;
            break;
          }
        }
      }
      else
        hs.add(c);
    }
    System.out.println(hs);
  }
```

```
} ///:~
```

The other way is almost certainly better:

1. This wasn't easy. I had to figure out the right **hashCode( )** to use, and as you can see above, I got snagged (twice!) on the '**==**' vs. '**.equals( )**' issue.

2. The fact that the only way to get an object back out of a **Set** is by hunting through the **Set** with an **Iterator** seems horribly inefficient compared to using a **Map**.

The output is:

```
[value: 19, occurrences: 4904 hashcode: 19
, value: 18, occurrences: 5034 hashcode: 18
, value: 17, occurrences: 5090 hashcode: 17
, value: 16, occurrences: 5035 hashcode: 16
, value: 15, occurrences: 4863 hashcode: 15
, value: 14, occurrences: 5023 hashcode: 14
, value: 13, occurrences: 5142 hashcode: 13
, value: 12, occurrences: 5073 hashcode: 12
, value: 11, occurrences: 5099 hashcode: 11
, value: 10, occurrences: 5062 hashcode: 10
, value: 9, occurrences: 5018 hashcode: 9
, value: 8, occurrences: 4985 hashcode: 8
, value: 7, occurrences: 4961 hashcode: 7
, value: 6, occurrences: 4907 hashcode: 6
, value: 5, occurrences: 5053 hashcode: 5
, value: 4, occurrences: 4966 hashcode: 4
, value: 3, occurrences: 4929 hashcode: 3
, value: 2, occurrences: 4869 hashcode: 2
, value: 1, occurrences: 5046 hashcode: 1
, value: 0, occurrences: 4941 hashcode: 0
]
```

Exercise 30

```
//: c11:E30_MapOrder2.java
/****************** Exercise 30 ************************
 * Fill a LinkedHashMap with String keys and objects of
 * your choice. Now extract the pairs, sort them based on
 * the keys, and re-insert them into the Map.
 *****************************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E30_MapOrder2 {
  public static void main(String[] args) {
    Map m1 = new LinkedHashMap();
    Collections2.fill(m1, Collections2.geography, 25);
    System.out.println(m1);
    Object keys[] = m1.keySet().toArray();
    Arrays.sort(keys);
    Map m2 = new LinkedHashMap();
    for(int i = 0; i < keys.length; i++)
      m2.put(keys[i], m1.get(keys[i]));
    System.out.println(m2);
  }
} ///:~
```

Exercise 31

```
//: c11:E31_HashedComparable.java
/****************** Exercise 31 *****************
 * Modify the class in Exercise 16 so that it
 * will work with HashSets and as a key in
 * HashMaps.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

class TwoString2 implements Comparable {
  String s1, s2;
  public TwoString2(String s1i, String s2i) {
    s1 = s1i;
    s2 = s2i;
  }
  public String toString() {
    return "\n[s1 = "+ s1 + ", s2 = " + s2 + "]";
  }
  public int compareTo(Object rv) {
    String rvi = ((TwoString2)rv).s1;
    return s1.compareTo(rvi);
  }
  public int hashCode() {
    // Since the comparisons are based on s1,
    // use s1's hashCode:
    return s1.hashCode();
  }
  public boolean equals(Object obj) {
    return obj instanceof TwoString2 &&
      ((TwoString2)obj).s1.equals(s1);
  }
  private static Arrays2.RandStringGenerator gen =
    new Arrays2.RandStringGenerator(7);
  public static Generator generator() {
    return new Generator() {
      public Object next() {
        return new TwoString2(
          (String)gen.next(),(String)gen.next());
      }
    };
  }
}

public class E31_HashedComparable {
  public static void main(String args[]) {
    HashSet hs = new HashSet();
    HashMap hm = new HashMap();
    Generator gen = TwoString2.generator();
    Collections2.fill(hs, gen , 20);
    for(int i = 0; i < 20; i++)
      hm.put(gen.next(), new Integer(i));
    System.out.println("hs = " + hs);
    System.out.println("hm = " + hm);
  }
} ///:~
```

You need to add **hashCode( )** and **equals( )** in order to use a class as a key in a hashed container. Since everything is based on **s1**, I built those methods also based on **s1**.

Here's the output from one run:

```
hs = [
[s1 = FdjoKjO, s2 = TBBelmN],
[s1 = AYHKzKM, s2 = BvruQhx],
[s1 = GPhTvyM, s2 = RAEFPyi],
[s1 = kSxNKgi, s2 = YIZzfHr],
[s1 = OWLOoJK, s2 = KpijXVc],
[s1 = VpuIkDq, s2 = PoLqKjr],
[s1 = ZrpLOIn, s2 = XeuUUmL],
[s1 = inMNRoc, s2 = iMYVIfI],
[s1 = oOJonQa, s2 = wZBAasY],
[s1 = bICiCiS, s2 = vZvKllo],
[s1 = xDFpfWm, s2 = GkONsnU],
[s1 = UbMGdqj, s2 = AtUOGRN],
[s1 = nPOHhUi, s2 = yiSigSC],
[s1 = LdNekTY, s2 = euVUWKS],
[s1 = JNeTerj, s2 = hPOCnln],
[s1 = CrFQntm, s2 = NqDAScD],
[s1 = oHewPvx, s2 = cdJuqZl],
[s1 = jVQGMIf, s2 = LHgNFwK],
[s1 = RDljQmR, s2 = SJWnetg],
[s1 = UnnfXYI, s2 = YvejakS]]
hm = {
[s1 = ZnFPLez, s2 = NVhnvaW]=3,
[s1 = EWZOvzb, s2 = TMzGxsE]=8,
[s1 = UeughJD, s2 = FYQEvuv]=5,
[s1 = qmUuGpy, s2 = MMBGNrM]=17,
[s1 = UMwSGzx, s2 = ntBqEgg]=0,
[s1 = vtrHlCB, s2 = gtttSpQ]=16,
[s1 = ovgVYLs, s2 = YePVJbD]=10,
[s1 = JPrHqnG, s2 = sPgYlAL]=9,
[s1 = MGPAawW, s2 = iQGvqfe]=7,
[s1 = NFgrEXy, s2 = AJBQGjm]=11,
[s1 = HfutkfJ, s2 = hAywUkY]=6,
[s1 = eDHLYBZ, s2 = QvWvUXi]=2,
[s1 = jllclgX, s2 = AFKOLXM]=12,
[s1 = eIOTsMc, s2 = MWNXUQJ]=15,
[s1 = BMcWAFy, s2 = yHHZmoH]=19,
[s1 = TWSpvza, s2 = NlcZQfK]=4,
[s1 = BWkbyxo, s2 = rhFYoMG]=13,
[s1 = xGuyIhq, s2 = VvKcEva]=1,
[s1 = Wbygvnf, s2 = NzClpCN]=14,
[s1 = CNZZqgd, s2 = nsMYoxh]=18}
```

Exercise 32

```
//: c11:E32_SlowSet.java
/****************** Exercise 32 *****************
 * Using SlowMap.java for inspiration, create a
 * SlowSet.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

class SlowSet extends AbstractSet {
  private ArrayList keys = new ArrayList();
  public boolean add(Object key) {
    if(!contains(key)) {
      keys.add(key);
      return true;
    } else
      return false;
  }
  public boolean contains(Object key) {
    return keys.contains(key);
  }
  public Iterator iterator() { return keys.iterator(); }
  public int size() { return keys.size(); }
}

public class E32_SlowSet {
  public static void main(String[] args) {
    SlowSet ss = new SlowSet();
    Collections2.fill(ss,
      Collections2.countries.reset(), 10);
    Collections2.fill(ss,
      Collections2.countries.reset(), 10);
    Collections2.fill(ss,
      Collections2.countries.reset(), 10);
    System.out.println(ss);
  }
} ///:~
```

I implemented the bare minimum to make it work, and tested it by repeatedly adding the same elements to the set. The output is:

```
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC, CHAD]
```

Exercise 33

```
//: c11:E33_ListPerformance2.java
// {RunByHand} (Takes too long during the build process)
/******************** Exercise 33 ***********************
 * Create a FastTraversalLinkedList that internally uses a
 * LinkedList for rapid insertions and removals, and an
 * ArrayList for rapid traversals and get() operations.
 * Test it by modifying ListPerformance.java.
 *********************************************************/
import com.bruceeckel.util.*;
import java.util.*;

class FastTraversalLinkedList extends AbstractList {
  private class FlaggedArrayList {
    private FlaggedLinkedList companion;
   boolean changed = false;
   private ArrayList list = new ArrayList();
   public void addCompanion(FlaggedLinkedList other) {
     companion = other;
   }
   private void synchronize() {
     if(companion.changed) {
       list = new ArrayList(companion.list);
       companion.changed = false;
     }
   }
    public Object get(int index) {
      synchronize();
      return list.get(index);
    }
    public Object remove(int index) {
```

```
    synchronize();
    changed = true;
    return list.remove(index);
  }
  public boolean remove(Object item) {
    synchronize();
    changed = true;
    return list.remove(item);
  }
  public int size() {
    synchronize();
    return list.size();
  }
  public Iterator iterator() {
    return list.iterator();
  }
}
private class FlaggedLinkedList {
  private FlaggedArrayList companion;
 boolean changed = false;
 LinkedList list = new LinkedList();
 public void addCompanion(FlaggedArrayList other) {
   companion = other;
 }
 private void synchronize() {
   if(companion.changed) {
     list = new LinkedList(companion.list);
     companion.changed = false;
   }
 }
  public void add(int index, Object item) {
    synchronize();
    changed = true;
    list.add(index, item);
  }
  public boolean add(Object item) {
    synchronize();
    changed = true;
    return list.add(item);
  }
}
```

```
  private FlaggedArrayList aList =
    new FlaggedArrayList();
  private FlaggedLinkedList lList =
    new FlaggedLinkedList();
  // Connect the two so they can synchronize:
  {
    aList.addCompanion(lList);
    lList.addCompanion(aList);
  }
  public int size() { return aList.size(); }
  public Object get(int arg) { return aList.get(arg); }
  public void add(int index, Object item) {
    lList.add(index, item);
  }
  public boolean add(Object item) {
    return lList.add(item);
  }
  // Through testing, we discover that the ArrayList is
  // actually much faster for removals than the LinkedList:
  public Object remove(int index) {
    return aList.remove(index);
  }
  public boolean remove(Object item) {
    return aList.remove(item);
  }
  public Iterator iterator() {
    return aList.iterator();
  }
}

public class E33_ListPerformance2 {
  private static int reps = 10000;
  private static int quantity = reps / 10;
  private abstract static class Tester {
    private String name;
    Tester(String name) { this.name = name; }
    abstract void test(List a);
  }
  private static Tester[] tests = {
    new Tester("get") {
      void test(List a) {
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < quantity; j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration") {
      void test(List a) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next(); // Produces value
        }
      }
    },
    // Modified so it doesn't use the iterator:
    new Tester("insert") {
      void test(List a) {
        int half = a.size()/2;
        String s = "test";
        for(int i = 0; i < reps * 10; i++)
          a.add(half, s);
      }
    },
    // Modified so it doesn't use the iterator:
    new Tester("remove at location") {
      void test(List a) {
        int half = a.size()/2;
        for(int loc = a.size()/2; loc < a.size(); loc++)
          a.remove(loc);
      }
    },
    new Tester("remove by object") {
      void test(List a) {
        int half = a.size()/2;
        for(int loc = a.size()/2; loc < a.size(); loc++) {
          Object item = a.get(loc);
          a.remove(item);
        }
      }
    },
  };
  public static void test(List a) {
    // Strip qualifiers from class name:
    System.out.println("Testing " +
      a.getClass().getName().replaceAll("\\w+\\.", ""));
    for(int i = 0; i < tests.length; i++) {
      Collections2.fill(a, Collections2.countries.reset(),
        quantity);
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(a);
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
    }
  }
  public static void main(String[] args) {
    // Choose a different number of
    // repetitions via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
    test(new ArrayList());
    test(new LinkedList());
    test(new FastTraversalLinkedList());
  }
} ///:~
```

To simplify the process of keeping the lists synchronized with each other, each type of internal list (**FlaggedArrayList** and **FlaggedLinkedList**) knows about the other, so when you perform an operation on one, it checks to see if the other has changed. If so, it updates itself from the other list, clears the other list's **changed** flag, and sets its own **changed** flag if it makes any modifications. You can see the two different lists connected to each other in the instance initialization clause.

Only the methods that are actually used in the tests have been implemented in **FlaggedArrayList** and **FlaggedLinkedList**. The **FastTraversalLinkedList** forwards method calls to the appropriate list depending on which one is more efficient.

The **tests** list has been modified from that in the book, so that the insertion and removal tests don't use the iterator (which would confuse the results). Also, "remove at location" and "remove by object" are now distinguished. The results of one run are as follows:

```
10000 repetitions
Testing ArrayList
get: 234
iteration: 1000
insert: 7578
remove at location: 907
remove by object: 2421
Testing LinkedList
get: 9047
iteration: 750
insert: 1813
remove at location: 18328
remove by object: 20922
Testing FastTraversalLinkedList
get: 469
iteration: 547
insert: 1500
remove at location: 906
remove by object: 2390
```

Surprisingly, removing an element in the middle of an **ArrayList** turns out to be far cheaper than removing an element from the middle of a **LinkedList**. Iteration is roughly the same, also somewhat surprising. In fact, the only place where the **LinkedList** is noticeably faster is insertions, and even that is only about 4 times faster.

Of course, more extensive testing would be required to ensure that the application where the **FastTraversalLinkedList** was used did not suffer excessively from overhead due to synchronization – if you constantly insert and then remove you will probably see unacceptable impacts due to list copying.

Exercise 34

```
//: c11:E34_SlowMapTest.java
```

```
/****************** Exercise 34 *****************
 * Apply the tests in Map1.java to SlowMap to
 * verify that it works. Fix anything in SlowMap
 * that doesn't work correctly.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

class SlowMap2 extends AbstractMap {
  private ArrayList
    keys = new ArrayList(),
    values = new ArrayList();
  public Object put(Object key, Object value) {
    Object result = get(key);
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
      values.set(keys.indexOf(key), value);
    return result;
  }
  public Object get(Object key) {
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  public Set entrySet() {
    Set entries = new HashSet();
    Iterator
      ki = keys.iterator(),
      vi = values.iterator();
    while(ki.hasNext())
      entries.add(new MPair(ki.next(), vi.next()));
    return entries;
  }
}

public class E34_SlowMapTest {
  static Collections2.StringPairGenerator geo =
    Collections2.geography;
  static Collections2.RandStringPairGenerator
```

```
    rsp = Collections2.rsp;
  // Producing a Set of the keys:
  public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() +", ");
    System.out.print("Keys: ");
    System.out.println(m.keySet());
  }
  // Producing a Collection of the values:
  public static void printValues(Map m) {
    System.out.print("Values: ");
    System.out.println(m.values());
  }
  public static void test(Map m) {
    Collections2.fill(m, geo, 25);
    // Map has 'Set' behavior for keys:
    Collections2.fill(m, geo.reset(), 25);
    printKeys(m);
    printValues(m);
    System.out.println(m);
    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[4][1];
    System.out.println("m.containsKey(\"" + key +
      "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): "
      + m.get(key));
    System.out.println("m.containsValue(\""
      + value + "\"): " + m.containsValue(value));
    Map m2 = new TreeMap();
    Collections2.fill(m2, rsp, 25);
    m.putAll(m2);
    printKeys(m);
    key = m.keySet().iterator().next().toString();
    System.out.println("First key in map: " + key);
    m.remove(key);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): " + m.isEmpty());
    Collections2.fill(m, geo.reset(), 25);
    // Operations on the Set change the Map:
    m.keySet().removeAll(m.keySet());
    System.out.println("m.isEmpty(): " + m.isEmpty());
  }
  public static void main(String[] args) {
    System.out.println("Testing SlowMap2");
    test(new SlowMap2());
  }
} ///:~
```

This was basically a copy and paste operation. Nothing was broken in **SlowMap**.

Here is **MPair.java**, duplicated for compilation purposes:

```
//: c11:MPair.java
import java.util.*;

public class MPair implements Map.Entry, Comparable {
  Object key, value;
  MPair(Object k, Object v) {
    key = k;
    value = v;
  }
  public Object getKey() { return key; }
  public Object getValue() { return value; }
  public Object setValue(Object v) {
    Object result = value;
    value = v;
    return result;
  }
  public boolean equals(Object o) {
    return key.equals(((MPair)o).key);
  }
  public int compareTo(Object rv) {
    return ((Comparable)key).compareTo(((MPair)rv).key);
  }
} ///:~
```

Exercise 35

```
//: c11:E35_FullSlowMap.java
/****************** Exercise 35 *****************
 * Implement the rest of the Map interface for
 * SlowMap.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

public class E35_FullSlowMap extends AbstractMap {
  private ArrayList
    keys = new ArrayList(),
    values = new ArrayList();
  public Object put(Object key, Object value) {
    Object result = get(key);
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
      values.set(keys.indexOf(key), value);
    return result;
  }
  public Object get(Object key) {
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  public Set entrySet() {
    Set entries = new HashSet();
    Iterator
      ki = keys.iterator(),
      vi = values.iterator();
    while(ki.hasNext())
      entries.add(new MPair(ki.next(), vi.next()));
    return entries;
  }
  public int size() { return keys.size(); }
  public boolean isEmpty() { return keys.isEmpty(); }
  public boolean containsValue(Object value) {
    return values.contains(value);
  }
  public boolean containsKey(Object key) {
    return keys.contains(key);
  }
  public Object remove(Object key) {
    if(!containsKey(key)) return null;
    int index = keys.indexOf(key);
    Object value = values.get(index);
    values.remove(index);
    keys.remove(index);
    return value;
  }
  public void putAll(Map t) {
    Iterator it = t.entrySet().iterator();
    while(it.hasNext()) {
      Entry me = (Entry)it.next();
      put(me.getKey(), me.getValue());
    }
    // Or you can just call super.putAll(t),
    // since it is implemented in AbstractMap.
  }
  public void clear() {
    keys.clear();
    values.clear();
  }
  public Set keySet() { return new HashSet(keys); }
  public Collection values() { return values; }
  public static void main(String[] args) {
    E35_FullSlowMap m = new E35_FullSlowMap();
    Collections2.fill(m, Collections2.geography, 25);
    E35_FullSlowMap m2 = new E35_FullSlowMap();
    m2.putAll(m);
    System.out.println("m2.entrySet() = " + m2.entrySet());
  }
} ///:~
```

Exercise 36

```
//: c11:E36_SlowMapPerformance.java
// {RunByHand}
/****************** Exercise 36 *****************
 * Modify MapPerformance.java to include tests of
 * SlowMap.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.*;

public class E36_SlowMapPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Map m, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("put") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++) {
          m.clear();
          Collections2.fill(m,
            Collections2.geography.reset(), size);
        }
      }
    },
    new Tester("get") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < size; j++)
            m.get(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = m.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  test(Map m, int size, int reps) {
    System.out.println("Testing " +
      m.getClass().getName() + " size " + size);
    Collections2.fill(m,
      Collections2.geography.reset(), size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(m, size, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Small:
    test(new TreeMap(), 10, reps);
    test(new HashMap(), 10, reps);
    test(new Hashtable(), 10, reps);
    test(new E35_FullSlowMap(), 10, reps);
    // Medium:
    test(new TreeMap(), 100, reps);
    test(new HashMap(), 100, reps);
    test(new Hashtable(), 100, reps);
    test(new E35_FullSlowMap(), 100, reps);
    // Large:
    test(new TreeMap(), 1000, reps);
    test(new HashMap(), 1000, reps);
    test(new Hashtable(), 1000, reps);
  }
} ///:~
```

This is a very simple copy and paste, with a few **test( )** lines added in **main( )** for **FullSlowMap**. The output is:

```
Testing java.util.TreeMap size 10
put: 50.0
get: 44.0
iteration: 60.0
Testing java.util.HashMap size 10
put: 28.0
get: 33.0
```

```
iteration: 71.0
Testing java.util.Hashtable size 10
put: 28.0
get: 32.0
iteration: 99.0
Testing FullSlowMap size 10
put: 38.0
get: 50.0
iteration: 587.0
Testing java.util.TreeMap size 100
put: 66.0
get: 51.6
iteration: 73.0
Testing java.util.HashMap size 100
put: 26.4
get: 40.1
iteration: 76.3
Testing java.util.Hashtable size 100
put: 26.4
get: 39.0
iteration: 80.2
Testing FullSlowMap size 100
put: 225.2
get: 221.9
iteration: 549.8
Testing java.util.TreeMap size 1000
put: 51.63
get: 64.48
iteration: 12.96
Testing java.util.HashMap size 1000
put: 17.85
get: 44.38
iteration: 13.3
Testing java.util.Hashtable size 1000
put: 18.78
get: 48.45
iteration: 15.81
```

You can see that iteration, in particular, is a horribly slow operation, and that **FullSlowMap** performance degrades badly with size. After about 10 minutes the ‘size 1000’ test still hadn’t completed for FullSlowMap so I took that one out. **SlowMap** clearly lives up to its name.

Exercise 37

```
//: c11:E37_MPairSlowMap.java
/****************** Exercise 37 *****************
 * Modify SlowMap so that instead of two
 * ArrayLists, it holds a single ArrayList of
 * MPair objects. Verify that the modified
 * version works correctly. Using
 * MapPerformance.java, test the speed of your
 * new Map. Now change the put() method so that
 * it performs a sort() after each pair is
 * entered, and modify get() to use
 * Collections.binarySearch() to look up the key.
 * Compare the performance of the new version
 * with the old ones.
 ***********************************************/
import com.bruceeckel.util.Collections2;
import java.util.*;

class SlowMap3 extends AbstractMap {
  private ArrayList mpairs = new ArrayList();
  public Object put(Object key, Object value) {
    MPair pair = new MPair(key, value);
    if(!mpairs.contains(pair))
      mpairs.add(pair);
    else
      mpairs.set(mpairs.indexOf(pair), pair);
    return pair;
  }
  public Object get(Object key) {
    MPair searchKey = new MPair(key, null);
    if(!mpairs.contains(searchKey))
      return null;
    return mpairs.get(mpairs.indexOf(searchKey));
  }
  public Set entrySet() {
    return new HashSet(mpairs);
    }
    public int size() {
      return mpairs.size();
    }
//  public boolean isEmpty() {
//    return mpairs.isEmpty();
//  }
//  public boolean containsKey(Object key) {
//    MPair pair = new MPair(key, null);
//    return mpairs.contains(key);
//  }
//  public Object remove(Object key) {
//    if(!containsKey(key)) return null;
//    MPair pair = new MPair(key, null);
//    int index = mpairs.indexOf(pair);
//    Object value =
//      ((MPair)mpairs.get(index)).getValue();
//    mpairs.remove(index);
//    return value;
//  }
//  public void putAll(Map t) {
//    Iterator it = t.entrySet().iterator();
//    while(it.hasNext()) {
//      Map.Entry me = (Map.Entry)it.next();
//      put(me.getKey(), me.getValue());
//    }
//  }
//  public void clear() {
//    mpairs.clear();
//  }
//  public Set keySet() {
//    Set ks = new HashSet();
//    Iterator it = mpairs.iterator();
//    while(it.hasNext())
//      ks.add(((MPair)it.next()).getKey());
//    return ks;
//  }
//  public Collection values() {
//    Collection vc = new ArrayList();
//    Iterator it = mpairs.iterator();
//    while(it.hasNext())
//      vc.add(((MPair)it.next()).getValue());
//    return vc;
//  }
}

public class E37_MPairSlowMap {
  public static void main(String args[]) {
    SlowMap3 sm3 = new SlowMap3();
    Collections2.fill(sm3, Collections2.geography, 25);
    System.out.println(sm3);
    System.out.println("Testing SlowMap3");
    E34_SlowMapTest.test(new SlowMap3());
  }
} ///:~
```

To test the new version of the **SlowMap**, you can just use the **test( )** method defined in **E34_SlowMapTest.java**.

In order to pass the tests, you only need the four methods defined above, but I've also included the commented methods necessary to fill out the class.

Exercise 38

```
//: c11:E38_CountedString.java
/****************** Exercise 38 *****************
 * Add a char field to CountedString that is also
 * initialized in the constructor, and modify the
 * hashCode() and equals() methods to include the
 * value of this char.
 ***********************************************/
import java.util.*;

class CountedString {
  private String s;
  private char c;
  private int id = 0;
  private static ArrayList created = new ArrayList();
  public CountedString(String str, char ci) {
    s = str;
    c = ci;
    created.add(s);
    Iterator it = created.iterator();
    // Id is the total number of instances
    // of this string in use by CountedString:
    while(it.hasNext())
      if(it.next().equals(s))
        id++;
  }
  public String toString() {
    return "\nString: " + s + " id: " + id +
      " char: " + c + " hashCode(): " + hashCode();
  }
  public int hashCode() { return s.hashCode() * id * c; }
  public boolean equals(Object o) {
    return (o instanceof CountedString)
      && s.equals(((CountedString)o).s)
      && c == ((CountedString)o).c
      && id == ((CountedString)o).id;
  }
}

public class E38_CountedString {
  public static void main(String[] args) {
    HashMap m = new HashMap();
    CountedString[] cs = new CountedString[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi", 'c');
      m.put(cs[i], new Integer(i));
    }
    System.out.println(m);
    for(int i = 0; i < cs.length; i++) {
      System.out.print("Looking up " + cs[i] + "\n");
      System.out.println(m.get(cs[i]));
    }
  }
} ///:~
```

You can see the addition of **char c** in **toString( )**, **hashCode( )** and **equals( )**. The output is:

```
{
String: hi id: 5 char: c hashCode(): 1647855=4,
String: hi id: 10 char: c hashCode(): 3295710=9,
String: hi id: 4 char: c hashCode(): 1318284=3,
String: hi id: 9 char: c hashCode(): 2966139=8,
String: hi id: 3 char: c hashCode(): 988713=2,
String: hi id: 8 char: c hashCode(): 2636568=7,
String: hi id: 2 char: c hashCode(): 659142=1,
String: hi id: 7 char: c hashCode(): 2306997=6,
String: hi id: 1 char: c hashCode(): 329571=0,
String: hi id: 6 char: c hashCode(): 1977426=5}
Looking up
String: hi id: 1 char: c hashCode(): 329571
0
Looking up
String: hi id: 2 char: c hashCode(): 659142
1
Looking up
String: hi id: 3 char: c hashCode(): 988713
2
Looking up
String: hi id: 4 char: c hashCode(): 1318284
3
Looking up
String: hi id: 5 char: c hashCode(): 1647855
4
Looking up
String: hi id: 6 char: c hashCode(): 1977426
5
Looking up
String: hi id: 7 char: c hashCode(): 2306997
6
Looking up
String: hi id: 8 char: c hashCode(): 2636568
7
Looking up
String: hi id: 9 char: c hashCode(): 2966139
8
Looking up
String: hi id: 10 char: c hashCode(): 3295710
9
```

Exercise 39

```
//: c11:E39_SimpleHashMapCollisons.java
/****************** Exercise 39 *****************
 * Modify SimpleHashMap so that it reports
 * collisions, and test this by adding the same
 * data set twice so that you see collisions.
 ***********************************************/
import com.bruceeckel.util.Collections2;
import java.util.*;

class SimpleHashMap2 extends AbstractMap {
  private final static int SZ = 997;
  private LinkedList[] bucket= new LinkedList[SZ];
  public Object put(Object key, Object value) {
    // Key-value pair to add:
    MPair pair = new MPair(key, value);
    Object result = null;
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    // Lines added here:
    else {
      System.out.println(
        "Collision while adding\n" + pair
        + "\nBucket already contains:");
      Iterator it = bucket[index].iterator();
      while(it.hasNext())
        System.out.println(it.next());
    }
    // End of lines added
    LinkedList pairs = bucket[index];
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Replace old with new
```

```
          found = true;
          break;
        }
      }
      if(!found)
        bucket[index].add(pair);
      return result;
    }
    public Object get(Object key) {
      int index = key.hashCode() % SZ;
      if(index < 0) index = -index;
      if(bucket[index] == null) return null;
      LinkedList pairs = bucket[index];
      MPair match = new MPair(key, null);
      ListIterator it = pairs.listIterator();
      while(it.hasNext()) {
        Object iPair = it.next();
        if(iPair.equals(match))
          return ((MPair)iPair).getValue();
      }
      return null;
    }
    public Set entrySet() {
      Set entries = new HashSet();
      for(int i = 0; i < bucket.length; i++) {
        if(bucket[i] == null) continue;
        Iterator it = bucket[i].iterator();
        while(it.hasNext())
          entries.add(it.next());
      }
      return entries;
    }
  }

  public class E39_SimpleHashMapCollisons {
    public static void main(String[] args) {
      SimpleHashMap2 m = new SimpleHashMap2();
      Collections2.fill(m, Collections2.geography, 25);
      Collections2.fill(m,
        Collections2.geography.reset(), 25);
      System.out.println(m);
  }
} ///:~
```

I added the lines surrounded by comments in the **put( )** method, and also moved the creation of the **pair** object up so that it could be printed when the collision occurred. Everything else in the **SimpleHashMap** stayed the same.

The output is:

```
Collision while adding
key = GUINEA value = -
Bucket already contains:
key = GUINEA value = Conakry
Collision while adding
key = ALGERIA value = Algiers
Bucket already contains:
key = ALGERIA value = Algiers
Collision while adding
key = ANGOLA value = Luanda
Bucket already contains:
key = ANGOLA value = Luanda
Collision while adding
key = BENIN value = Porto-Novo
Bucket already contains:
key = BENIN value = Porto-Novo
Collision while adding
key = BOTSWANA value = Gaberone
Bucket already contains:
key = BOTSWANA value = Gaberone
Collision while adding
key = BURKINA FASO value = Ouagadougou
Bucket already contains:
key = BURKINA FASO value = Ouagadougou
Collision while adding
key = BURUNDI value = Bujumbura
Bucket already contains:
key = BURUNDI value = Bujumbura
Collision while adding
key = CAMEROON value = Yaounde
Bucket already contains:
key = CAMEROON value = Yaounde
```

```
Collision while adding
key = CAPE VERDE value = Praia
Bucket already contains:
key = CAPE VERDE value = Praia
Collision while adding
key = CENTRAL AFRICAN REPUBLIC value = Bangui
Bucket already contains:
key = CENTRAL AFRICAN REPUBLIC value = Bangui
Collision while adding
key = CHAD value = N'djamena
Bucket already contains:
key = CHAD value = N'djamena
Collision while adding
key = COMOROS value = Moroni
Bucket already contains:
key = COMOROS value = Moroni
Collision while adding
key = CONGO value = Brazzaville
Bucket already contains:
key = CONGO value = Brazzaville
Collision while adding
key = DJIBOUTI value = Dijibouti
Bucket already contains:
key = DJIBOUTI value = Dijibouti
Collision while adding
key = EGYPT value = Cairo
Bucket already contains:
key = EGYPT value = Cairo
Collision while adding
key = EQUATORIAL GUINEA value = Malabo
Bucket already contains:
key = EQUATORIAL GUINEA value = Malabo
Collision while adding
key = ERITREA value = Asmara
Bucket already contains:
key = ERITREA value = Asmara
Collision while adding
key = ETHIOPIA value = Addis Ababa
Bucket already contains:
key = ETHIOPIA value = Addis Ababa
Collision while adding
```

```
key = GABON value = Libreville
Bucket already contains:
key = GABON value = Libreville
Collision while adding
key = THE GAMBIA value = Banjul
Bucket already contains:
key = THE GAMBIA value = Banjul
Collision while adding
key = GHANA value = Accra
Bucket already contains:
key = GHANA value = Accra
Collision while adding
key = GUINEA value = Conakry
Bucket already contains:
key = GUINEA value = -
Collision while adding
key = GUINEA value = -
Bucket already contains:
key = GUINEA value = Conakry
Collision while adding
key = BISSAU value = Bissau
Bucket already contains:
key = BISSAU value = Bissau
Collision while adding
key = CETE D'IVOIR (IVORY COAST)
value = Yamoussoukro
Bucket already contains:
key = CETE D'IVOIR (IVORY COAST)
value = Yamoussoukro
Collision while adding
key = KENYA value = Nairobi
Bucket already contains:
key = KENYA value = Nairobi
{ANGOLA=Luanda, CAMEROON=Yaounde, GUINEA=-,
BURUNDI=Bujumbura, GHANA=Accra, EGYPT=Cairo,
KENYA=Nairobi, BURKINA FASO=Ouagadougou,
BISSAU=Bissau, ETHIOPIA=Addis Ababa,
BOTSWANA=Gaberone, BENIN=Porto-Novo,
GABON=Libreville, ERITREA=Asmara, CENTRAL AFRICAN
REPUBLIC=Bangui, DJIBOUTI=Dijibouti,
CAPE VERDE=Praia, THE GAMBIA=Banjul,
ALGERIA=Algiers, EQUATORIAL GUINEA=Malabo,
CONGO=Brazzaville, COMOROS=Moroni,
CHAD=N'djamena,
CETE D'IVOIR (IVORY COAST)=Yamoussoukro}
```

Exercise 40

```
//: c11:E40_LeftToReader.java
/****************** Exercise 40 *****************
 * Modify SimpleHashMap so that it reports the
 * number of "probes" necessary when collisions
 * occur. That is, how many calls to next() must
 * be made on the Iterators that walk the
 * LinkedLists looking for matches?
 ***********************************************/
public class E40_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Modify the **get( )** method in the same way that the previous exercise modified **put( )**.

Exercise 41

```
//: c11:E41_SimpleHashMapClearRemove.java
/****************** Exercise 41 *****************
 * Implement the clear() and remove() methods for
 * SimpleHashMap.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

class SimpleHashMap3 extends AbstractMap {
  // Choose a prime number for the hash table
  // size, to achieve a uniform distribution:
  private final static int SZ = 997;
  private LinkedList[] bucket= new LinkedList[SZ];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    LinkedList pairs = bucket[index];
    MPair pair = new MPair(key, value);
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Replace old with new
        found = true;
        break;
      }
    }
    if(!found)
      bucket[index].add(pair);
    return result;
  }
  public Object get(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match))
        return ((MPair)iPair).getValue();
    }
    return null;
  }
  public Set entrySet() {
    Set entries = new HashSet();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        entries.add(it.next());
    }
    return entries;
  }
  public void clear() {
    // Effectively erase everything by allocating
    // a new empty array of buckets:
    bucket= new LinkedList[SZ];
  }
  public Object remove(Object key) {
    // Code is copied from get(), except that
    // before returning the value, the MPair is
    // removed from the list:
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match)) {
        // Changes are here:
        Object value = ((MPair)iPair).getValue();
        // Removes the last fetched value:
        it.remove();
        return value;
      }
    }
    return null;
  }
}

public class E41_SimpleHashMapClearRemove {
  public static void main(String[] args) {
    SimpleHashMap3 m = new SimpleHashMap3();
    Collections2.fill(m, Collections2.geography, 10);
    System.out.println(m);
    System.out.println("m.get(\"BURUNDI\") = "
      + m.get("BURUNDI"));
    m.remove("BURUNDI");
    System.out.println(
      "After removal, m.get(\"BURUNDI\") = "
      + m.get("BURUNDI"));
    m.clear();
    System.out.println("After clearing: " + m);
  }
} ///:~
```

See comments embedded in **clear( )** and **remove( )**, above.

The output is:

```
{CAMEROON=Yaounde, BENIN=Porto-Novo,
ALGERIA=Algiers, CENTRAL AFRICAN REPUBLIC=Bangui,
BOTSWANA=Gaberone, BURKINA FASO=Ouagadougou,
ANGOLA=Luanda, CHAD=N'djamena, BURUNDI=Bujumbura,
CAPE VERDE=Praia}
m.get("BURUNDI") = Bujumbura
After removal, m.get("BURUNDI") = null
After clearing: {}
```

Exercise 42

```
//: c11:E42_FullSimpleHashMap.java
/****************** Exercise 42 *****************
 * Implement the rest of the Map interface for
 * SimpleHashMap.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

class FullSimpleHashMap extends AbstractMap {
  // Choose a prime number for the hash table
  // size, to achieve a uniform distribution:
  private final static int SZ = 997;
  private LinkedList[] bucket= new LinkedList[SZ];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    LinkedList pairs = bucket[index];
    MPair pair = new MPair(key, value);
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Replace old with new
        found = true;
        break;
      }
    }
    if(!found)
      bucket[index].add(pair);
    return result;
  }
  public Object get(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match))
        return ((MPair)iPair).getValue();
    }
    return null;
  }
  public Set entrySet() {
    Set entries = new HashSet();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        entries.add(it.next());
    }
    return entries;
  }
  public void clear() {
    // Effectively erase everything by allocating
    // a new empty array of buckets:
    bucket= new LinkedList[SZ];
  }
  public Object remove(Object key) {
    // Code is copied from get(), except that
    // before returning the value, the MPair is
    // removed from the list:
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match)) {
        // Changes are here:
        Object value = ((MPair)iPair).getValue();
        // Removes the last fetched value:
        it.remove();
        return value;
      }
    }
    return null;
  }
  public int size() {
    int sz = 0;
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      sz++;
    }
    return sz;
  }
  public boolean isEmpty() {
    // Could just say return size() == 0;
    // but this is faster:
    for(int i = 0; i < bucket.length; i++)
```

```
      if(bucket[i] == null) continue;
      else return false;
    return true;
  }
  public boolean containsValue(Object value) {
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        if(((MPair)it.next()).getValue().equals(value))
          return true;
    }
    return false;
  }
  public boolean containsKey(Object key) {
    // A slight modification of
    // the previous method:
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        if(((MPair)it.next()).getKey().equals(key))
          return true;
    }
    return false;
  }
  public void putAll(Map t) {
    // AbstractMap has a working implementation:
    super.putAll(t);
  }
  public Set keySet() {
    // A variation of entrySet():
    Set keys = new HashSet();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        keys.add(((MPair)it.next()).getKey());
    }
    return keys;
  }
  public Collection values() {
    // A variation of keySet():
    Collection values = new ArrayList();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        values.add(((MPair)it.next()).getValue());
    }
    return values;
  }
  public boolean equals(Object o) {
    // Version in AbstractMap is OK:
    return super.equals(o);
  }
}

public class E42_FullSimpleHashMap {
  public static void main(String args[]) {
    FullSimpleHashMap m = new FullSimpleHashMap(),
    m2 = new FullSimpleHashMap();
    Collections2.fill(m, Collections2.geography, 10);
    Collections2.fill(m2,
      Collections2.geography.reset(), 10);
    System.out.println("m.size() = " + m.size());
    System.out.println("m.isEmpty() = " + m.isEmpty());
    System.out.println("m.equals(m2) = " + m.equals(m2));
    System.out.println("m.keySet() = " + m.keySet());
    System.out.println("m.values() = " + m.values());
  }
} ///:~
```

As you can see above, it's important to realize that sometimes the methods are already implemented correctly in the abstract base version of the container you're implementing (not that you implement your own containers all that often!).

Here's the output. Some methods have been tested in previous examples.

```
m.size() = 10
m.isEmpty() = false
m.equals(m2) = true
```

```
m.keySet() = [BOTSWANA, CENTRAL AFRICAN REPUBLIC,
BURKINA FASO, CAPE VERDE, ANGOLA, BENIN, CHAD,
BURUNDI, CAMEROON, ALGERIA]
m.values() = [Gaberone, Porto-Novo, Algiers,
Ouagadougou, Yaounde, Bangui, Praia, N'djamena,
Luanda, Bujumbura]
```

Exercise 43

```
//: c11:E43_SimpleHashMapRehash.java
/****************** Exercise 43 *****************
 * Add a private rehash() method to SimpleHashMap
 * that is invoked when the load factor exceeds
 * 0.75. During rehashing, double the number of
 * buckets, then search for the first prime
 * number greater than that to determine the new
 * number of buckets.
 ***********************************************/
import java.util.*;
import com.bruceeckel.util.*;

class SimpleHashMap4 extends AbstractMap {
  private int count = 0; // Number of elements
  private static final double loadFactor = 0.75;
  // Use the same initial capacity as in
  // java.util.HashMap:
  private final static int initialCapacity = 11;
  private int capacity = initialCapacity;
  private int threshold = (int)(capacity * loadFactor);
  private LinkedList[] bucket = new LinkedList[capacity];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode() % capacity;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    LinkedList pairs = bucket[index];
    MPair pair = new MPair(key, value);
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Replace old with new
        found = true;
        break;
      }
    }
    if(!found) {
      if(count >= threshold)
        rehash();
      if(bucket[index] == null)
        bucket[index] = new LinkedList();
      bucket[index].add(pair);
      count++;
    }
    return result;
  }
  public Object get(Object key) {
    int index = key.hashCode() % capacity;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match))
        return ((MPair)iPair).getValue();
    }
    return null;
  }
  public Set entrySet() {
    Set entries = new HashSet();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        entries.add(it.next());
    }
    return entries;
  }
  private boolean isPrime(int candidate) {
    for(int j = 2; j < candidate; j++)
      if(candidate % j == 0) return false;
    return true;
  }
  private int nextPrime(int candidate) {
    while(!isPrime(candidate))
      candidate++;
    return candidate;
  }
  private void rehash() {
    // Points to a new Set of the entries, so it
    // won't be bothered by what we're about to do:
    Iterator it = entrySet().iterator();
    // Get next prime capacity:
    capacity = nextPrime(capacity * 2);
    System.out.println(
      "Rehashing, new capacity = " + capacity);
    bucket = new LinkedList[capacity];
    threshold = (int)(capacity * loadFactor);
    // Fill new buckets (crude, but it works):
    while(it.hasNext()) {
      MPair mp = (MPair)it.next();
      put(mp.getKey(), mp.getValue());
    }
  }
}

public class E43_SimpleHashMapRehash {
  public static void main(String[] args) {
    SimpleHashMap4 m = new SimpleHashMap4();
    Collections2.fill(m, Collections2.geography, 50);
    System.out.println(m);
  }
} ///:~
```

This solution is “the simplest thing that could possibly work,” so efficiency is not taken into consideration here. Some of the information was gleaned by studying the source code from **java.util.HashMap** which is distributed in the JDK.

At the top of the class, you'll see a number of new fields have been added: **count,** to keep track of the number of elements, **loadFactor**, which in this case is fixed at 0.75 (which is the default for **HashMap**), the **initialCapacity** of 11 (also the same as **HashMap**), the **capacity**, which is the current number of buckets (called **SZ** in **SimpleHashMap.java** in the book) and the **threshold**, which determines when to call **rehash( )**.

In **put( )**, you can see the code that is added to call **rehash( )**. The other methods are left alone, and at the bottom of the class there are three new methods: **isPrime( )** and **nextPrime( )** for prime number generation, and **rehash( )** itself.

The first thing that **rehash( )** does is get an **Iterator** to the **entrySet( )** (a set of all keys and values) for this map. If you look up at the code for **entrySet( )** you'll see that it makes a new collection to hold the objects. Because of this, it won't be affected when we start changing the **bucket** container, and there's no chance the objects will be garbage collected while we make those changes.

The first thing to do is increase the number of buckets to a new prime number that is more than double the current number (as specified by the exercise). The **capacity** is doubled, then given to **nextPrime( )**. All **nextPrime( )** does is call **isPrime( )** to see if the number is prime, and if not increments the **candidate** and tries it again. Note that this is not a very efficient way to do things, and if you look at the **java.util.HashMap** source code you'll see it just doubles the size and adds one, which is sometimes prime but usually not. As a further exercise, create a lookup table for prime numbers to be used in the map.

A new **bucket** array is created with the new **capacity**, the new **threshold** is calculated, and the **entrySet( )** iterator is used along with **put( )** to reload the elements into the hash table. Again, this is not the most efficient way to do things, but it works. You should look at the **java.util.HashMap** source code for a more efficient approach.

The output is:

```
Rehashing, new capacity = 23
```

```
Rehashing, new capacity = 47
Rehashing, new capacity = 97
Rehashing, new capacity = 197
{CETE D'IVOIR (IVORY COAST)=Yamoussoukro, MALAWI=Lilongwe,
DJIBOUTI=Dijibouti,
MAURITIUS=Port Louis, THE GAMBIA=Banjul,
ANGOLA=Luanda, SEYCHELLES=Victoria, TUNISIA=Tunis,
GUINEA=-, NIGER=Niamey, CAMEROON=Yaounde,
SENEGAL=Dakar, MAURITANIA=Nouakchott,
SIERRA LEONE=Freetown, ALGERIA=Algiers,
MOZAMBIQUE=Maputo, NAMIBIA=Windhoek,
CAPE VERDE=Praia, GHANA=Accra, COMOROS=Moroni,
BURKINA FASO=Ouagadougou, RWANDA=Kigali,
BOTSWANA=Gaberone, EGYPT=Cairo, MALI=Bamako,
ETHIOPIA=Addis Ababa, CHAD=N'djamena,
SUDAN=Khartoum, BENIN=Porto-Novo, ERITREA=Asmara,
CENTRAL AFRICAN REPUBLIC=Bangui,
BURUNDI=Bujumbura, LESOTHO=Maseru,
MADAGASCAR=Antananarivo, TOGO=Lome,
SOMALIA=Mogadishu, MOROCCO=Rabat,
SOUTH AFRICA=Pretoria/Cape Town, LIBYA=Tripoli,
EQUATORIAL GUINEA=Malabo, SWAZILAND=Mbabane,
TANZANIA=Dodoma, BISSAU=Bissau, GABON=Libreville,
LIBERIA=Monrovia, KENYA=Nairobi,
CONGO=Brazzaville, SAO TOME E PRINCIPE=Sao Tome,
NIGERIA=Abuja}
```

Exercise 44

```
//: c11:E44_SimpleHashSet.java
/****************** Exercise 44 *****************
 * Following the example in SimpleHashMap.java,
 * create and test a SimpleHashSet.
 ***********************************************/
import com.bruceeckel.util.Collections2;
import java.util.*;

class SimpleHashSet extends AbstractSet {
  // Choose a prime number for the hash table
  // size, to achieve a uniform distribution:
```

```
  private final static int SZ = 997;
  private LinkedList[] bucket= new LinkedList[SZ];
  public boolean add(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    ListIterator it = bucket[index].listIterator();
    while(it.hasNext())
      if(it.next().equals(key))
        return false;
    // Sets do not permit duplicates and one
    // was already in the set.
    it.add(key);
    return true; // Successful add
  }
  public boolean contains(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return false;
    Iterator it = bucket[index].iterator();
    while(it.hasNext())
      if(it.next().equals(key))
        return true;
    return false;
  }
  public int size() {
    int sz = 0;
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        it.next();
        sz++;
    }
    return sz;
  }
  public Iterator iterator() {
    ArrayList al = new ArrayList();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        al.add(it.next());
    }
    return al.iterator();
  }
}

public class E44_SimpleHashSet {
  public static void main(String[] args) {
    SimpleHashSet m = new SimpleHashSet();
    Collections2.fill(m, Collections2.countries, 10);
    Collections2.fill(m,
      Collections2.countries.reset(), 10);
    System.out.println("m = " + m);
    System.out.println("m.size() = " + m.size());
    Iterator it = m.iterator();
    System.out.println("it.next()= "+ it.next());
  }
} ///:~
```

This only contains keys, and not values. Also, different methods are declared in **Set** and implemented in **AbstractSet**, and the above methods are the only ones absolutely necessary to implement the **SimpleHashSet**. The output is:

```
m = [BOTSWANA, BENIN, ALGERIA, BURKINA FASO,
CAMEROON, CENTRAL AFRICAN REPUBLIC, CAPE VERDE,
CHAD, ANGOLA, BURUNDI]
m.size() = 10
it.next() = BOTSWANA
```

Exercise 45

```
//: c11:E45_SimpleHashMapArrays.java
// {RunByHand}
/****************** Exercise 45 *****************
 * Modify SimpleHashMap to use ArrayLists instead
 * of LinkedLists. Modify MapPerformance.java to
 * compare the performance of the two
 * implementations.
 *********************************************/
import java.util.*;
import com.bruceeckel.util.*;

class SimpleHashMap5 extends AbstractMap {
  private final static int SZ = 997;
  private ArrayList[] bucket= new ArrayList[SZ];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new ArrayList();
    ArrayList pairs = bucket[index];
    MPair pair = new MPair(key, value);
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Replace old with new
        found = true;
        break;
      }
    }
    if(!found)
      bucket[index].add(pair);
    return result;
  }
  public Object get(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    ArrayList pairs = bucket[index];
    MPair match = new MPair(key, null);
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match))
        return ((MPair)iPair).getValue();
    }
    return null;
  }
  public Set entrySet() {
    Set entries = new HashSet();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        entries.add(it.next());
    }
    return entries;
  }
}

public class E45_SimpleHashMapArrays {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Map m, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("put") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++) {
          m.clear();
          Collections2.fill(m,
            Collections2.geography.reset(), size);
        }
      }
    },
    new Tester("get") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < size; j++)
            m.get(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = m.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void test(Map m, int size, int reps) {
    System.out.println("Testing " +
      m.getClass().getName() + " size " + size);
    Collections2.fill(m,
      Collections2.geography.reset(), size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(m, size, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    SimpleHashMap5 m = new SimpleHashMap5();
    Collections2.fill(m, Collections2.geography, 25);
    System.out.println(m);
    int reps = 50000;
    // Small:
    test(new FullSimpleHashMap(), 10, reps);
    test(new SimpleHashMap5(), 10, reps);
    // Medium:
    test(new FullSimpleHashMap(), 100, reps);
    test(new SimpleHashMap5(), 100, reps);
    // Large:
    test(new FullSimpleHashMap(), 1000, reps);
    test(new SimpleHashMap5(), 1000, reps);
  }
} ///:~
```

This is copy and paste with a few small edits. The comparison yields:

```
{THE GAMBIA=Banjul, EQUATORIAL GUINEA=Malabo,
KENYA=Nairobi, ERITREA=Asmara, DJIBOUTI=Dijibouti,
BURUNDI=Bujumbura, GABON=Libreville,
ALGERIA=Algiers, CAMEROON=Yaounde, COMOROS=Moroni,
ANGOLA=Luanda, CETE D'IVOIR (IVORY COAST)=
Yamoussoukro, CONGO=Brazzaville, BISSAU=Bissau,
BOTSWANA=Gaberone, CAPE VERDE=Praia,
CHAD=N'djamena, GHANA=Accra, BURKINA
FASO=Ouagadougou, CENTRAL AFRICAN REPUBLIC=Bangui,
GUINEA=-, ETHIOPIA=Addis Ababa, EGYPT=Cairo,
BENIN=Porto-Novo}
Testing FullSimpleHashMap size 10
put: 165.0
get: 38.0
iteration: 1395.0
Testing SimpleHashMap5 size 10
put: 181.0
get: 44.0
iteration: 1302.0
Testing FullSimpleHashMap size 100
put: 76.3
get: 40.7
iteration: 663.5
Testing SimpleHashMap5 size 100
put: 115.3
get: 43.4
iteration: 652.0
Testing FullSimpleHashMap size 1000
put: 61.02
get: 53.44
iteration: 115.13
Testing SimpleHashMap5 size 1000
put: 62.23
get: 51.52
iteration: 105.29
```

The linked list version is more efficient, but this is probably because an iterator is used to traverse the **ArrayList** instead of random-access indexing, which is most likely more efficient in that case. Try changing **SimpleHashMap5** to use random-access indexing for **get( )** and see if that makes it more efficient.

The JDK comes with source code – try looking at the source for **HashMap** and see the way the chaining is implemented there. They manage their own singly-linked list, which is most likely a lot more efficient than using **List** objects designed for general-purpose use.

Exercise 46

```
//: c11:E46_HashMapLoadFactor.java
// {RunByHand}
/****************** Exercise 46 *****************
 * Using the HTML documentation for the JDK
 * (downloadable from java.sun.com), look up the
 * HashMap class. Create a HashMap, fill it with
 * elements, and determine the load factor. Test
 * the lookup speed with this map, then attempt
 * to increase the speed by making a new HashMap
 * with a larger initial capacity and copying the
 * old map into the new one, running your lookup
 * speed test again on the new map.
 ***********************************************/
import com.bruceeckel.util.*;
import java.util.HashMap;

public class E46_HashMapLoadFactor {
  public static
  void testGet(HashMap filledMap, int n) {
    for(int tc = 0; tc < 10000000; tc++)
      for(int i = 0; i < n; i++) {
        Object key = CountryCapitals.pairs[i][0];
        if(filledMap.get(key) == null)
          System.out.println("Not found: " + key);
      }
  }
  public static void main(String args[]) {
    // Initial capacity 11:
    HashMap map = new HashMap();
    // Fill to less than threshold:
    Collections2.fill(map,
      Collections2.geography.reset(), 8);
    // Initial capacity 22:
```

```
    HashMap map2 = new HashMap(22);
    // Fill to less than threshold:
    Collections2.fill(map2,
      Collections2.geography.reset(), 16);
    long t1 = System.currentTimeMillis();
    testGet(map, 8);
    long t2 = System.currentTimeMillis();
    System.out.println("map : " + (t2 - t1));
    t1 = System.currentTimeMillis();
    testGet(map2, 8);
    t2 = System.currentTimeMillis();
    System.out.println("map2 : " + (t2 - t1));
  }
} ///:~
```

From the book:

Load factor*:* size/capacity. A load factor of 0 is an empty table, 0.5 is a half-full table, etc.

Since there are no **public** methods to access the capacity etc. information, you must calculate it by knowing the necessary information in the map. There are two ways to do this: establish it yourself by calling the appropriate **HashMap** constructor, or look at the source code that comes with the JDK to see what is done by default. Taking the latter approach, we see that the default constructor has initial values of:

initialCapacity = 11

loadFactor = .75

We also see the **threshold** in the source code. This is the value at which the **rehash( )** method is called to reorganize the table. From the JDK source:

```
threshold = (int)(initialCapacity * loadFactor);
```

Which means the table will be increased when the number of elements in the table are greater than (int)(0.75 * 11) = 8. So when the ninth element is added, the table will be rehashed.

You can make a **HashMap** with a larger initial capacity by using the constructor that takes one argument: **HashMap(int initialCapacity)**. So the line:

```
HashMap map2 = new HashMap(22);
```

produces a **HashMap** with double the default initial capacity, and the threshold is (int)(0.75 * 22) = 16.

The output from one run is:

```
map : 10390
map2 : 9440
```

The larger map does cost a bit more for lookups, it would appear.

Exercise 47

```
//: c11:E47_GreenhouseLinkedList.java
/****************** Exercise 47 *****************
 * In Chapter 8, locate the GreenhouseController.java
 * example, which consists of four files. In
 * Controller.java, the class Controller uses an ArrayList.
 * Change the code to use a LinkedList instead, and use an
 * Iterator to cycle through the set of events.
 ***********************************************/
import java.util.*;

class Controller {
  // List changed to a LinkedList:
  private List eventList = new LinkedList();
  private List pending = new LinkedList();
  // Cannot add events to the eventList as it's being
  // iterated through because it produces a
  // ConcurrentModificationException:
  public void addEvent(Event c) { pending.add(c); }
  // After each iteration pass, any pending events
  // are appended:
  private void addPendingEvents() {
    if(pending.size() > 0) {
      eventList.addAll(pending);
      pending.clear();
    }
  }
  public void run() {
    addPendingEvents();
    while(eventList.size() > 0) {
      ListIterator it = eventList.listIterator();
      while(it.hasNext()) {
        Event e = (Event)it.next();
        if(e.ready()) {
          System.out.println(e);
          e.action();
          it.remove();
        }
      }
      addPendingEvents();
    }
  }
}

abstract class Event {
  private long eventTime;
  protected final long delayTime;
  public Event(long delayTime) {
    this.delayTime = delayTime;
    start();
  }
  public void start() { // Allows restarting
    eventTime = System.currentTimeMillis() + delayTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= eventTime;
  }
  public abstract void action();
}

class GreenhouseControls extends Controller {
  private boolean light = false;
  public class LightOn extends Event {
    public LightOn(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String toString() { return "Light is on"; }
  }
  public class LightOff extends Event {
    public LightOff(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String toString() { return "Light is off"; }
  }
  private boolean water = false;
  public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here.
      water = true;
    }
    public String toString() {
      return "Greenhouse water is on";
    }
  }
  public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
      // Put hardware control code here.
      water = false;
    }
    public String toString() {
      return "Greenhouse water is off";
    }
  }
  private String thermostat = "Day";
  public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here.
      thermostat = "Night";
    }
    public String toString() {
      return "Thermostat on night setting";
    }
  }
  public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here.
      thermostat = "Day";
    }
    public String toString() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
      addEvent(new Bell(delayTime));
    }
    public String toString() { return "Bing!"; }
  }
  public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
      super(delayTime);
      this.eventList = eventList;
      for(int i = 0; i < eventList.length; i++)
        addEvent(eventList[i]);
    }
    public void action() {
      for(int i = 0; i < eventList.length; i++) {
        eventList[i].start(); // Rerun each event
        addEvent(eventList[i]);
      }
      start(); // Rerun this Event
      addEvent(this);
    }
    public String toString() {
      return "Restarting system";
    }
  }
  public class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Terminating";  }
  }
}

public class E47_GreenhouseLinkedList {
  public static void main(String[] args) {
    GreenhouseControls gc = new GreenhouseControls();
    // Instead of hard-wiring, you could parse
    // configuration information from a text file here:
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
      gc.new LightOn(200),
      gc.new LightOff(400),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    // Stop after 8 seconds:
    gc.addEvent(gc.new Terminate(8000));
    gc.run();
  }
} ///:~
```

In this solution, all the classes are simply copied from the book and collected into a single file, but **Controller** is the only class that is actually modified.

As you can see in the comments in **Controller**, using an iterator with a single list became problematic, because the overridden **action( )**

methods can add new **Event**s to the end of the list – with a single list, this modified the list during iteration and produced a **ConcurrentModificationException**. The problem was solved by modifying **addEvent()** to add **Event**s to a second list, and then calling **synchronize()** to add any pending **Event**s back into the main list when the iteration was complete.

Exercise 48

```
//: c11:E48_LeftToReader.java
/****************** Exercise 48 *****************
 * (Challenging). Write your own hashed map
 * class, customized for a particular key type:
 * String for this example. Do not inherit it
 * from Map. Instead, duplicate the methods so
 * that the put() and get() methods specifically
 * take String objects, not Objects, as keys.
 * Everything that involves keys should not use
 * generic types, but instead work with Strings,
 * to avoid the cost of upcasting and
 * downcasting. Your goal is to make the fastest
 * possible custom implementation. Modify
 * MapPerformance.java to test your
 * implementation vs. a HashMap.
 ***********************************************/
public class E48_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Exercise 49

```
//: c11:E49_LeftToReader.java
/****************** Exercise 49 *****************
 * (Challenging). Find the source code for List
 * in the Java source code library that comes
 * with all Java distributions. Copy this code
 * and make a special version called intList that
 * holds only ints. Consider what it would take
 * to make a special version of List for all the
 * primitive types. Now consider what happens if
 * you want to make a linked list class that
 * works with all the primitive types.
 * Parameterized types in Java JDK 1.5 provide a
 * way to do this work for you automatically.
 ***********************************************/
public class E49_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Exercise left to reader");
  }
} ///:~
```

Exercise 50

```
//: c11:E50_Month2Comparable.java
/****************** Exercise 50 *****************
 * Modify c08:Month.java to make it implement
 * the Comparable interface.
 ***********************************************/
import java.util.*;

final class Month2 implements Comparable {
  private String name;
  private int order;
  private Month2(int ord, String nm) {
    order = ord;
    name = nm;
  }
  public String toString() { return name; }
  public final static Month2
    JAN = new Month2(1, "January"),
    FEB = new Month2(2, "February"),
    MAR = new Month2(3, "March"),
    APR = new Month2(4, "April"),
    MAY = new Month2(5, "May"),
    JUN = new Month2(6, "June"),
    JUL = new Month2(7, "July"),
    AUG = new Month2(8, "August"),
    SEP = new Month2(9, "September"),
    OCT = new Month2(10, "October"),
    NOV = new Month2(11, "November"),
    DEC = new Month2(12, "December");
  public final static Month2[] month =  {
    JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public final static Month2 number(int ord) {
    return month[ord - 1];
  }
  public int compareTo(Object rv) {
    int argOrder = ((Month2)rv).order;
    return (order < argOrder ? -1 :
      (order == argOrder ? 0 : 1));
  }
}

public class E50_Month2Comparable {
  public static void main(String[] args) {
    List months = new ArrayList();
    for(int i = 0; i < 20; i++)
      months.add(Month2.month[(int)(Math.random()
        * Month2.month.length)]);
    System.out.println("months = " + months);
    Collections.sort(months);
    System.out.println("after sorting, months = "
      + months);
  }
} ///:~
```

The comparison is not based on alphabetical ordering of the months, but rather time ordering. In **main( )**, a **List** is filled with a random assortment of **Month3** objects, printed, and then sorted. The output from one run is:

```
months = [June, June, April, December, March,
October, December, August, December, May, April,
February, June, October, March, July, February,
December, March, May]
after sorting, months = [February, February,
March, March, March, April, April, May, May,
June, June, June, July, August, October, October,
December, December, December, December]
```

Exercise 51

```
//: c11:E51_CountedString2.java
/********************** Exercise 51 **********************
 * Modify the hashCode() in CountedString.java by removing
 * the combination with id, and demonstrate that
 * CountedString still works as a key. What is the problem
 * with this approach?
 *********************************************************/
import java.util.*;

public class E51_CountedString2 {
  private static List created = new ArrayList();
  private String s;
  private int id = 0;
  public E51_CountedString2(String str) {
    s = str;
    created.add(s);
    Iterator it = created.iterator();
    // id is the total number of instances
    // of this string in use by E51_CountedString2:
    while(it.hasNext())
      if(it.next().equals(s))
        id++;
  }
  public String toString() {
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode();
  }
  public int hashCode() {
    // The very simple approach:
    // return s.hashCode() * id;
    // Using Joshua Bloch's recipe:
    int result = 17;
    result = 37*result + s.hashCode();
    // result = 37*result + id;
    return result;
  }
  public boolean equals(Object o) {
    return (o instanceof E51_CountedString2)
      && s.equals(((E51_CountedString2)o).s)
      && id == ((E51_CountedString2)o).id;
  }
  public static void main(String[] args) {
    Map map = new HashMap();
    E51_CountedString2[] cs = new E51_CountedString2[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new E51_CountedString2("hi");
      map.put(cs[i], new Integer(i));
    }
    System.out.println(map);
    for(int i = 0; i < cs.length; i++) {
      System.out.println("Looking up " + cs[i]);
      System.out.println(map.get(cs[i]));
    }
  }
} ///:~
```

The only change to the original program is the commenting of the line:

```
    // result = 37*result + id;
```

Without that comment, we see in the output (edited for clarity):

```
Looking up String: hi id: 1 hashCode(): 146447
Looking up String: hi id: 2 hashCode(): 146448
Looking up String: hi id: 3 hashCode(): 146449
Looking up String: hi id: 4 hashCode(): 146450
Looking up String: hi id: 5 hashCode(): 146451
Looking up String: hi id: 6 hashCode(): 146452
Looking up String: hi id: 7 hashCode(): 146453
Looking up String: hi id: 8 hashCode(): 146454
Looking up String: hi id: 9 hashCode(): 146455
Looking up String: hi id: 10 hashCode(): 146456
```

Each object clearly has a unique hashcode. With the line above commented, the edited output is:

```
Looking up String: hi id: 1 hashCode(): 3958
Looking up String: hi id: 2 hashCode(): 3958
Looking up String: hi id: 3 hashCode(): 3958
Looking up String: hi id: 4 hashCode(): 3958
Looking up String: hi id: 5 hashCode(): 3958
Looking up String: hi id: 6 hashCode(): 3958
Looking up String: hi id: 7 hashCode(): 3958
Looking up String: hi id: 8 hashCode(): 3958
Looking up String: hi id: 9 hashCode(): 3958
Looking up String: hi id: 10 hashCode(): 3958
```

The lookup still produces the correct values, but each object hashes to the same value – all the values fall in the same bucket. Thus the performance advantage of hashing is lost because **equals( )** must be used to compare the values until the desired object is found.

Chapter 12

Exercise 1

```
//: c12:E01_FileIntoList.java
/****************** Exercise 1 ******************
 * Open a text file so that you can read the file
 * one line at a time. Read each line as a String
 * and place that String object into a
 * LinkedList. Print all of the lines in the
 * LinkedList in reverse order.
 ***********************************************/
import java.io.*;
import java.util.*;

public class E01_FileIntoList {
  // Report all exceptions to console:
  public static void main(String args[])
  throws Exception {
    LinkedList lines = new LinkedList();
    BufferedReader in =
      new BufferedReader(
        new FileReader("E01_FileIntoList.java"));
    String s;
    while((s = in.readLine())!= null) lines.add(s);
    in.close();
    ListIterator it = lines.listIterator(lines.size());
    while(it.hasPrevious())
      System.out.println(it.previous());
  }
} ///:~
```

The output is the above file in reversed line order.

Exercise 2

```
//: c12:E02_CommandLine.java
```

```
// {Args: E02_CommandLine.java}
/****************** Exercise 2 ******************
 * Modify Exercise 1 so that the name of the file
 * you read is provided as a command-line
 * argument.
 ***********************************************/
import java.util.*;
import java.io.*;

public class E02_CommandLine {
  // Report all exceptions to console:
  public static void main(String args[])
  throws Exception {
    LinkedList lines = new LinkedList();
    BufferedReader in =
      new BufferedReader(
        new FileReader(args[0]));
    String s;
    while((s = in.readLine())!= null)
      lines.add(s);
    in.close();
    ListIterator it =
      lines.listIterator(lines.size());
    while(it.hasPrevious())
      System.out.println(it.previous());
  }
} ///:~
```

Exercise 3

```
//: c12:E03_LineNumber.java
// {Args: E03_LineNumber.java numbered.txt}
/****************** Exercise 3 ******************
 * Modify Exercise 2 to also open a text file so
 * you can write text into it. Write the lines in
 * the ArrayList, along with line numbers (do not
 * attempt to use the "LineNumber" classes), out
 * to the file.
 ***********************************************/
import java.util.*;
import java.io.*;

public class E03_LineNumber {
  // Report all exceptions to console:
  public static void main(String args[])
  throws Exception {
    LinkedList lines = new LinkedList();
    BufferedReader in =
      new BufferedReader(
        new FileReader(args[0]));
    String s;
    while((s = in.readLine())!= null)
      lines.add(s);
    in.close();
    PrintWriter out =
      new PrintWriter(
        new BufferedWriter(
          new FileWriter(args[1])));
    int line = 1;
    Iterator it = lines.iterator();
    while(it.hasNext())
      out.println(line++ + ": " + it.next());
    out.close();
  }
} ///:~
```

Exercise 4

```
//: c12:E04_UpperCase.java
// {Args: E04_UpperCase.java}
/****************** Exercise 4 ******************
 * Modify Exercise 2 to force all the lines in
 * the ArrayList to upper case and send the
 * results to System.out.
 ***********************************************/
import java.util.*;
import java.io.*;

public class E04_UpperCase {
  // Report all exceptions to console:
```

```
  public static void main(String args[])
  throws Exception {
    LinkedList lines = new LinkedList();
    BufferedReader in =
      new BufferedReader(
        new FileReader(args[0]));
    String s;
    while((s = in.readLine())!= null)
      lines.add(s);
    in.close();
    Iterator it = lines.iterator();
    while(it.hasNext())
      System.out.println(
        ((String)it.next()).toUpperCase());
  }
} ///:~
```

Exercise 5

```
//: c12:E05_FindWords.java
// {Args: E05_FindWords.java util main candidate args}
/****************** Exercise 5 ******************
 * Modify Exercise 2 to take additional
 * command-line arguments of words to find in the
 * file. Print all lines in which any of the
 * words match.
 ***********************************************/
import java.util.*;
import java.io.*;

public class E05_FindWords {
  // Report all exceptions to console:
  public static void main(String args[]) throws Exception {
    Set words = new HashSet();
    for(int i = 1; i < args.length; i++)
      words.add(args[i]);
    BufferedReader in = new BufferedReader(
      new FileReader(args[0]));
    String s;
    while((s = in.readLine())!= null) {
      Iterator it = words.iterator();
      while(it.hasNext()) {
        String candidate = (String)it.next();
        if(s.indexOf(candidate) != -1) {
          System.out.println(s);
          break; // Out of while loop
        }
      }
    }
  }
} ///:~
```

I use a **Set** here to automatically eliminate duplicate candidate words on the command line. The **break** statement prevents the line from being printed more than once, if more than one candidate word is present in the line.

Exercise 6

```
//: c12:E06_SearchWords.java
// {Args: java util main Reader}
/****************** Exercise 6 ******************
 * Modify DirList.java so that the FilenameFilter
 * actually opens each file and accepts the file
 * based on whether any of the trailing arguments
 * on the command line exist in that file.
 ***********************************************/
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

class DirFilter implements FilenameFilter {
  String afn;
  Set words = new HashSet();
  DirFilter(String afn, String[] args) {
    this.afn = afn;
    for(int i = 1; i < args.length; i++)
      words.add(args[i]);
  }
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    // Only investigate files containing afn:
    if(f.indexOf(afn) == -1) return false;
    System.out.println("** " + f);
    // Modified from E05_FindWords.java:
    try {
      BufferedReader in =
        new BufferedReader(
          new FileReader(f));
      String s;
      while((s = in.readLine())!= null) {
        Iterator it = words.iterator();
        while(it.hasNext()) {
          String candidate = (String)it.next();
          if(s.indexOf(candidate) != -1) {
            // Print the line that first matched:
            System.out.println(s);
            return true;
          }
        }
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return false;
  }
}

public class E06_SearchWords {
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    list = path.list(new DirFilter(args[0], args));
    Arrays.sort(list, new AlphabeticComparator());
    System.out.println(">> Resulting List <<");
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
} ///:~
```

Exercise 7

```
//: c12:E07_DirList4.java
// {Args: \\w+\.java}
/********************** Exercise 7 **********************
 * Modify DirList.java to produce all the file names in the
 * current directory and subdirectories that satisfy the
 * given regular expression. Hint: use recursion to
 * traverse the subdirectories
 * ******************************************************/
import java.io.*;
import java.util.*;
import java.util.regex.*;
import com.bruceeckel.util.*;

public class E07_DirList4 {
  private static void
  listDir(File dir, final String regEx, String indent) {
    String[] files = dir.list(new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regEx);
      public boolean accept(File dir, String name) {
        File f = new File(name);
        if(f.isDirectory()) return true;
        return pattern.matcher(f.getName()).matches();
      }
   });
   Arrays.sort(files, new AlphabeticComparator());
   for(int i = 0; i < files.length; i++) {
     System.out.println(indent + "+--" + files[i]);
     File child = new File(files[i]);
     if(child.isDirectory())
       listDir(child, regEx , indent + "   ");
    }
  }
  public static void main(final String[] args) {
    if(args.length < 1) {
      System.out.println("Argument required: <regex>");
      System.exit(1);
    }
    File f = new File(".");
    System.out.println(f.getName());
    listDir(f, args[0], "   ");
  }
} ///:~
```

Exercise 8

```
//: c12:E08_SortedDirList.java
/****************** Exercise 8 ******************
 * Create a class called SortedDirList with a
 * constructor that takes file path information
 * and builds a sorted directory list from the
 * files at that path. Create two overloaded
 * list() methods that will either produce the
 * whole list or a subset of the list based on an
 * argument. Add a size() method that takes a
 * file name and produces the size of that file.
 ***********************************************/
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

class SortedDirList {
  File path;
  public SortedDirList() {
    path = new File(".");
  }
  public SortedDirList(String filePath) {
    path = new File(filePath);
  }
  public String[] list() {
    String[] list =
      path.list(new FilenameFilter() {
        public boolean
        accept(File dir, String n) {
          // Accept anything:
          return true;
        }
      });
    Arrays.sort(list, new AlphabeticComparator());
    return list;
  }
  public String[] list(final String afn) {
    String[] list =
      path.list(new FilenameFilter() {
        public boolean
        accept(File dir, String n) {
          String f = new File(n).getName();
          return f.indexOf(afn) != -1;
        }
      });
    Arrays.sort(list, new AlphabeticComparator());
    return list;
  }
  public long length(final String fileName) {
    File[] files = path.listFiles(new FileFilter() {
      public boolean accept(File pathname) {
        return pathname.getName().indexOf(fileName) != -1;
      }
    });
    if(files.length > 1)
      throw new RuntimeException(
        "More than one match for " + fileName);
    return files[0].length();
  }
}

public class E08_SortedDirList {
  public static void main(String args[]) {
    // Default constructor == current directory
    SortedDirList dir = new SortedDirList();
    System.out.println(Arrays.asList(dir.list()));
    System.out.println(Arrays.asList(dir.list(".java")));
    String f = dir.list(".java")[0];
    System.out.println(
      "Length of " + f + " is " + dir.length(f));
  }
} ///:~
```

Much of this is a rewrite of **DirList.java** into a reusable class. However, the **length( )** method provided a different challenge because a **File** object doesn't act much like a container – for example, you can't just say "get me the file with this name." I looked up the **File** class in the JDK docs, and found **listFiles( )**, which returns an array of **File** objects given a **FileFilter** implementation (same format as **FilenameFilter**). The **FileFilter** is implemented as an anonymous inner class, and it just uses **getName( )** (which produces only the file name, not the path) to get the file name which it then searches for a match with the **fileName** you're looking for. Note that an exception is thrown if there is more than one match, which would indicate some kind of error. Then the **File.length( )** method produces the number of bytes in the file.

Here's the output from one run:

```
[build.xml, CVS, E01_FileIntoList.java,
E02_CommandLine.java, E03_LineNumber.java,
E04_UpperCase.java, E05_FindWords.java,
E06_SearchWords.java, E07_DirList4.java,
E08_SortedDirList.class, E08_SortedDirList.java,
E09_AlphabeticWordCount.java, E10_WordCountSet.java,
E11_CountLines.java, E12_BufferPerformance.java,
E13_StripSpaces.java, E14_RepairCADState.java,
E15_BlipCheck.java, E16_Blip3Test.java,
E17_GreenhouseConfig.txt, E17_GreenhouseConfigFile.java,
E18_PrintCharBuff.java, E19_AllocateDirect.java,
E20_CheckForMatch.java, E21_CheckForMatch2.java,
E22_JGrep2.java, E23_JGrep3.java, E24_JGrep4.java,
SortedDirList$1.class, SortedDirList$2.class,
SortedDirList$3.class, SortedDirList.class, temp.txt]
[E01_FileIntoList.java, E02_CommandLine.java,
E03_LineNumber.java, E04_UpperCase.java,
E05_FindWords.java, E06_SearchWords.java,
E07_DirList4.java, E08_SortedDirList.java,
E09_AlphabeticWordCount.java, E10_WordCountSet.java,
E11_CountLines.java, E12_BufferPerformance.java,
E13_StripSpaces.java, E14_RepairCADState.java,
E15_BlipCheck.java, E16_Blip3Test.java,
E17_GreenhouseConfigFile.java, E18_PrintCharBuff.java,
E19_AllocateDirect.java, E20_CheckForMatch.java,
E21_CheckForMatch2.java, E22_JGrep2.java, E23_JGrep3.java,
E24_JGrep4.java]
```

```
Length of E01_FileIntoList.java is 903
```

Exercise 9

```
//: c12:E09_AlphabeticWordCount.java
// {Args: E09_AlphabeticWordCount.java}
/****************** Exercise 9 ******************
 * Modify WordCount.java so that it produces an
 * alphabetic sort instead, using the tool from
 * Chapter 11.
 ***********************************************/
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

class Counter {
  private int i = 1;
  int read() { return i; }
  void increment() { i++; }
}

class WordCount {
  private FileReader file;
  private StreamTokenizer st;
  // The only change -- provide a comparator
  // to the TreeMap constructor:
  private TreeMap counts =
    new TreeMap(new AlphabeticComparator());
    // Instead of this:
    // new TreeMap();
  WordCount(String filename) {
    try {
      file = new FileReader(filename);
      st = new StreamTokenizer(
        new BufferedReader(file));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.err.println("Could not open " + filename);
      throw new RuntimeException(e);
    }
  }
  void dispose() {
    try {
      file.close();
    } catch(IOException e) {
      System.err.println("file.close() unsuccessful");
      throw new RuntimeException(e);
    }
  }
  void countWords() {
    try {
      while(st.nextToken() !=
        StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = st.sval; // Already a String
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
        if(counts.containsKey(s))
          ((Counter)counts.get(s)).increment();
        else
          counts.put(s, new Counter());
      }
    } catch(IOException e) {
      System.err.println("st.nextToken() unsuccessful");
    }
  }
  Collection values() {
    return counts.values();
  }
  Set keySet() { return counts.keySet(); }
  Counter getCounter(String s) {
    return (Counter)counts.get(s);
  }
}

public class E09_AlphabeticWordCount {
  public static void main(String[] args)
  throws FileNotFoundException {
    WordCount wc =
      new WordCount(args[0]);
    wc.countWords();
    Iterator keys = wc.keySet().iterator();
    while(keys.hasNext()) {
      String key = (String)keys.next();
      System.out.println(key + ": " +
        wc.getCounter(key).read());
    }
    wc.dispose();
  }
} ///:~
```

This is almost trivial. All you have to do is change the construction of the **TreeMap** so that it takes an **AlphabeticComparator**. Everything else is basically the same as in the original example.

Exercise 10

```
//: c12:E10_WordCountSet.java
// {Args: E10_WordCountSet.java}
/****************** Exercise 10 ******************
 * Modify WordCount.java so that it uses a class
 * containing a String and a count value to store
 * each different word, and a Set of these
 * objects to maintain the list of words.
 ***********************************************/
import java.io.*;
import java.util.*;

class Counter2 implements Comparable {
  private String word;
  private int count;
  public Counter2(String word) {
    this.word = word;
    count = 1;
  }
  public void increment() { count++; }
  public String toString() {
    return "\n" + word + " [" + count + "]";
  }
  public boolean equals(Object obj) {
    return obj instanceof Counter2 &&
      ((Counter2)obj).word.equals(word);
  }
  public int hashCode() {
    return word.hashCode();
  }
  public int compareTo(Object o) {
    return word.compareTo(((Counter2)o).word);
  }
}

class CounterSet extends AbstractSet {
  private Map set = new TreeMap();
  public void addOrIncrement(String s) {
    Counter2 c = new Counter2(s);
    if(set.containsKey(c))
      ((Counter2)set.get(c)).increment();
    else
      set.put(c, c);
  }
  public Iterator iterator() {
    return set.keySet().iterator();
  }
  public int size() {
    return set.size();
  }
  public String toString() {
    return set.keySet().toString();
  }
}

class WordCount2 {
  private FileReader file;
  private StreamTokenizer st;
  private CounterSet counts = new CounterSet();
  WordCount2(String filename) {
    try {
      file = new FileReader(filename);
      st = new StreamTokenizer(new BufferedReader(file));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
  void dispose() {
    try {
      file.close();
    } catch(IOException e) {
      System.err.println("file.close() unsuccessful");
      throw new RuntimeException(e);
    }
  }
  void countWords() {
    try {
      while(st.nextToken() !=
        StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = st.sval; // Already a String
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
        counts.addOrIncrement(s);
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public Iterator iterator() {
    return counts.iterator();
  }
  public String toString() {
    return counts.toString();
  }
}

public class E10_WordCountSet {
  public static void main(String[] args)
  throws FileNotFoundException {
    WordCount2 wc =  new WordCount2(args[0]);
    wc.countWords();
    System.out.println("wc = " + wc);
    wc.dispose();
  }
} ///:~
```

This turned out to be a bit challenging because **java.util.Set** does not have a way to fetch objects out of a **Set**, except by walking through with an iterator, which seems inelegant. However, that would have been one way to correctly solve the problem.

I decided to implement my own **Set** by using a classic trick – a **Map** where the keys and values are the same (this is often the way sets are implemented). The requirement that each key must be unique gives it set behavior, and now it's possible to use **get( )** to fetch objects back out of the set. The **CounterSet.addOrIncrement( )** method takes a string, creates a **Counter2** object (which must implement **Comparable** so that it can be placed inside a **TreeSet**), and if that **Counter2** doesn't already exist, places it in the set with a default count of 1 (since the first word was found). Otherwise, it uses **get( )** to fetch the existing **Counter2** and increment it.

Since **WordCount2** has a **toString( )** method it can be printed with **println( )**.

Exercise 11

```
//: c12:E11_CountLines.java
/****************** Exercise 11 *****************
 * Modify IOStreamDemo.java so that it uses
 * LineNumberReader to keep track of the
 * line count. Note that it's much easier to just
 * keep track programmatically.
 ***********************************************/
import java.io.*;

public class E11_CountLines {
  public static void main(String[] args)
  throws IOException {
    // LineNumberReader is inherited from
    // BufferedReader so we don't need to
    // explicitly buffer it:
    LineNumberReader in =
      new LineNumberReader(
        new FileReader("E11_CountLines.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += in.getLineNumber() + ": " + s + "\n";
    in.close();
    System.out.println(s2);
  }
} ///:~
```

Note that I changed the exercise to use the more modern **LineNumberReader**.

Here's the output:

```
1: //: c12:E11_CountLines.java
2: /****************** Exercise 11 *****************
3:  * Modify IOStreamDemo.java so that it uses
4:  * LineNumberReader to keep track of the
5:  * line count. Note that it's much easier to just
6:  * keep track programmatically.
7:  ***********************************************/
8: import java.io.*;
```

```
9:
10: public class E11_CountLines {
11:   public static void main(String[] args)
12:   throws IOException {
13:     // LineNumberReader is inherited from
14:     // BufferedReader so we don't need to
15:     // explicitly buffer it:
16:     LineNumberReader in =
17:       new LineNumberReader(
18:         new FileReader("E11_CountLines.java"));
19:     String s, s2 = new String();
20:     while((s = in.readLine())!= null)
21:       s2 += in.getLineNumber() + ": " + s + "\n";
22:     in.close();
23:     System.out.println(s2);
24:   }
25: } ///:~
```

LineNumberReader counts from one, interestingly enough. It makes sense, but most everything else counts from zero so it would have been hard to guess how it would come out.

Exercise 12

```
//: c12:E12_BufferPerformance.java
/****************** Exercise 12 *****************
 * Starting with section 4 of IOStreamDemo.java,
 * write a program that compares the performance
 * of writing to a file when using buffered and
 * unbuffered I/O.
 ***********************************************/
import java.io.*;

public class E12_BufferPerformance {
  // Report exceptions to console:
  public static void main(String args[])
  throws Exception {
    BufferedReader in = new BufferedReader(
      new FileReader("E12_BufferPerformance.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
```

```
      s2 += s + "\n";
    in.close();
    PrintWriter out1 = new PrintWriter(new BufferedWriter(
      new FileWriter("BufferPerformance1.out")));
    long t1 = System.currentTimeMillis();
    for(int i = 0; i < 10000; i++)
      out1.println(s2);
    long t2 = System.currentTimeMillis();
    System.out.println("buffered: " + (t2 - t1));
    out1.close();
    PrintWriter out2 = new PrintWriter(
      new FileWriter("BufferPerformance2.out"));
    t1 = System.currentTimeMillis();
    for(int i = 0; i < 10000; i++)
      out2.println(s2);
    t2 = System.currentTimeMillis();
    System.out.println("unbuffered: " + (t2 - t1));
    out2.close();
  }
} ///:~
```

The output is:

```
buffered: 2250
unbuffered: 4840
```

Exercise 13

```
//: c12:E13_StripSpaces.java
/****************** Exercise 13 *****************
 * Modify section 5 of IOStreamDemo.java to
 * eliminate the spaces in the line produced by
 * the first call to in5.readUTF().
 ***********************************************/
import java.io.*;

public class E13_StripSpaces {
  // Report all exceptions to console:
  public static void main(String args[])
  throws Exception {
    DataOutputStream out2 =
      new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Data.txt")));
    out2.writeDouble(3.14159);
    out2.writeUTF("That was pi");
    out2.writeDouble(1.41413);
    out2.writeUTF("Square root of 2");
    out2.close();
    DataInputStream in5 =
      new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("Data.txt")));
    System.out.println(in5.readDouble());
    // First, eliminate any surrounding space:
    String s = in5.readUTF().trim();
    String noSpace = "";
    char c;
    for(int i = 0; i < s.length(); i++) {
      c = s.charAt(i);
      if(c != ' ')
        noSpace += c;
    }
    System.out.println("noSpace = " + noSpace);
    System.out.println(in5.readDouble());
    System.out.println(in5.readUTF());
  }
} ///:~
```

This one was a problem, because the exercise was left over from the first edition of the book, so it referred to a variable that no longer existed. I modified the exercise description and made the best of it, but it ended up being about **String** manipulation and not IO, so it will be changed or dropped in the next edition of the book.

The output is:

```
3.14159
noSpace = Thatwaspi
1.41413
Square root of 2
```

Exercise 14

```
//: c12:E14_RepairCADState.java
// {Args: CADState.out}
/****************** Exercise 14 *****************
 * Repair the program CADState.java as described
 * in the text.
 ***********************************************/
/*
The instructions from the book are:
1. Add a serializeStaticState() and
    deserializeStaticState() to the shapes.
2. Remove the ArrayList shapeTypes and all code
    related to it.
3. Add calls to the new serialize and
    deserialize static methods in the shapes.
*/
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
  public static final int
    RED = 1, BLUE = 2, GREEN = 3;
  private int xPos, yPos, dimension;
  private static Random r = new Random();
  private static int counter = 0;
  abstract public void setColor(int newColor);
  abstract public int getColor();
  public Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      " color[" + getColor() +
      "] xPos[" + xPos +
      "] yPos[" + yPos +
      "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.nextInt() % 100;
    switch(counter++ % 3) {
      default:
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static int color = RED;
  public static void
  serializeStaticState(ObjectOutputStream os)
      throws IOException {
    os.writeInt(color);
  }
  public static void
  deserializeStaticState(ObjectInputStream os)
      throws IOException {
    color = os.readInt();
  }
  public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}

class Square extends Shape {
  private static int color = BLUE;
  public static void
  serializeStaticState(ObjectOutputStream os)
      throws IOException {
```

```
    os.writeInt(color);
  }
  public static void
  deserializeStaticState(ObjectInputStream os)
      throws IOException {
    color = os.readInt();
  }
  public Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = RED;
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}

class Line extends Shape {
  private static int color = GREEN;
  public static void
  serializeStaticState(ObjectOutputStream os)
      throws IOException {
    os.writeInt(color);
  }
  public static void
  deserializeStaticState(ObjectInputStream os)
      throws IOException {
    color = os.readInt();
  }
  public Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}
```

```
public class E14_RepairCADState {
  public static void main(String[] args) throws Exception {
    if(args.length < 1) {
      System.out.println("Argument required: filename");
      System.exit(1);
    }
    ArrayList shapes = new ArrayList();
    // Make some shapes:
    for(int i = 0; i < 10; i++)
      shapes.add(Shape.randomFactory());
    // Save the state vector:
    ObjectOutputStream out =
      new ObjectOutputStream(
        new FileOutputStream(args[0]));
    Line.serializeStaticState(out);
    Circle.serializeStaticState(out);
    Square.serializeStaticState(out);
    out.writeObject(shapes);
    out.close();
    // Now read the file back in:
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream(args[0]));
    // Read in the same order they were written:
    Line.deserializeStaticState(in);
    Circle.deserializeStaticState(in);
    Square.deserializeStaticState(in);
    shapes = (ArrayList)in.readObject();
    // Display the shapes:
    System.out.println(shapes);
  }
} ///:~
```

I've given each shape a different default color. **main()** now takes a command-line argument of a file name, and first writes the objects out to that file, then reads the file back in. The result of running the second (for one run) is:

```
[class Circle color[1] xPos[-30] yPos[-16] dim[84]
, class Square color[1] xPos[-2] yPos[2] dim[50]
, class Line color[3] xPos[35] yPos[-37] dim[37]
, class Circle color[1] xPos[29] yPos[-50] dim[99]
, class Square color[1] xPos[83] yPos[-25] dim[-92]
, class Line color[3] xPos[-7] yPos[-37] dim[-64]
, class Circle color[1] xPos[62] yPos[-39] dim[-50]
, class Square color[1] xPos[11] yPos[-18] dim[24]
, class Line color[3] xPos[34] yPos[44] dim[84]
, class Circle color[1] xPos[-83] yPos[-17] dim[43]
]
```

You can see that the stored color is retrieved for each type, rather than using the default values.

Exercise 15

```
//: c12:E15_BlipCheck.java
/****************** Exercise 15 *****************
 * In Blips.java, copy the file and rename it to
 * BlipCheck.java and rename the class Blip2 to
 * BlipCheck (making it public and removing the
 * public scope from the class Blips in the
 * process). Remove the //! marks in the file and
 * execute the program including the offending
 * lines. Next, comment out the default
 * constructor for BlipCheck. Run it and explain
 * why it works. Note that after compiling, you
 * must execute the program with "java Blips"
 * because the main() method is still in class
 * Blips.
 ***********************************************/
import java.io.*;

class Blip1 implements Externalizable {
  public Blip1() {
    System.out.println("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
```

```
      throws IOException, ClassNotFoundException {
    System.out.println("Blip1.readExternal");
  }
}

public class E15_BlipCheck
implements Externalizable {
//  E15_BlipCheck() {
//    System.out.println("BlipCheck Constructor");
//  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("BlipCheck.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("BlipCheck.readExternal");
  }
  public static void main(String[] args) throws Exception {
    // To make it run with Ant.
    Blips.main(args);
  }
}

class Blips {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip1 b1 = new Blip1();
    E15_BlipCheck b2 = new E15_BlipCheck();
    ObjectOutputStream o =
      new ObjectOutputStream(
        new FileOutputStream("Blips.out"));
    System.out.println("Saving objects:");
    o.writeObject(b1);
    o.writeObject(b2);
    o.close();
    // Now get them back:
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("Blips.out"));
    System.out.println("Recovering b1:");
    b1 = (Blip1)in.readObject();
    // OOPS! Throws an exception:
    System.out.println("Recovering b2:");
    b2 = (E15_BlipCheck)in.readObject();
  }
} ///:~
```

When the **//!'s** are removed, the output is:

```
Constructing objects:
Blip1 Constructor
BlipCheck Constructor
Saving objects:
Blip1.writeExternal
BlipCheck.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
Recovering b2:
java.io.InvalidClassException: E14_BlipCheck;
IllegalAccessException
  at
java.io.ObjectInputStream.inputObject(ObjectInputStream.jav
a:1224)
  at
java.io.ObjectInputStream.readObject(ObjectInputStream.java
:386)
  at
java.io.ObjectInputStream.readObject(ObjectInputStream.java
:236)
  at Blips.main(C:/aaa-TIJ2-
solutions/code/c11/E14_BlipCheck.java:71)
```

When the constructor is subsequently commented out (as in the listing above) the output is:

```
Constructing objects:
Blip1 Constructor
Saving objects:
Blip1.writeExternal
```

```
BlipCheck.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
Recovering b2:
BlipCheck.readExternal
```

The exception is removed because, by eliminating the explict default constructor, you give the compiler permission to generate the default constructor for you. The automatically-generated default constructor for a **public** class is made **public** by the compiler, and so it will work as an **Externalizable** object.

Exercise 16

```
//: c12:E16_Blip3Test.java
/****************** Exercise 16 *****************
 * In Blip3.java, comment out the two lines after
 * the phrases "You must do this:" and run the
 * program. Explain the result and why it differs
 * from when the two lines are in the program.
 ***********************************************/
import java.io.*;

class Blip3 implements Externalizable {
  int i;
  String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialized only in nondefault
    // constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
  throws IOException {
    System.out.println("Blip3.writeExternal");
    // You must do this:
//    out.writeObject(s);
//    out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // You must do this:
//    s = (String)in.readObject();
//    i =in.readInt();
  }
}

public class E16_Blip3Test {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3);
    ObjectOutputStream o =
      new ObjectOutputStream(
        new FileOutputStream("Blip3.out"));
    System.out.println("Saving object:");
    o.writeObject(b3);
    o.close();
    // Now get it back:
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("Blip3.out"));
    System.out.println("Recovering b3:");
    b3 = (Blip3)in.readObject();
    System.out.println(b3);
  }
} ///:~
```

Here's the output without commenting the lines:

```
Constructing objects:
Blip3(String x, int a)
A String 47
```

```
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
```

Here's the output when the lines are commented:

```
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
null0
```

The **writeExternal( )** method is called by **writeObject( )**, and the **readExternal( )** method is called by **readObject( )**. If these do not explicitly write and read the parts of the object that are of interest, they don't get stored or retrieved and the object only gets its default values for its fields, which in this case are **null** and **0**.

Exercise 17

```
//: c12:E17_GreenhouseConfigFile.java
// {Args: E17_GreenhouseConfig.txt}
/****************** Exercise 17 *****************************
 * (Intermediate) In Chapter 8, locate the
 * GreenhouseController.java example, which consists of
 * four files. GreenhouseController contains a hard-coded
 * set of events. Change the program so that it reads the
 * events and their relative times from a text file.
 * (Challenging: use a design patterns Factory Method to
 * build the events—see Thinking in Patterns (with Java) at
 * www.BruceEckel.com.)
 ***********************************************************/
package c12;
import java.util.*;
import java.io.*;
import java.lang.reflect.*;

abstract class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
}

class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
      throw new RuntimeException(
        "Tried to add too many events");
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() { events[next] = null; }
```

```
}

class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
      }
    }
  }
}

class GreenhouseControls extends Controller {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String description() {
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String description() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = true;
    }
    public String description() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = false;
    }
    public String description() {
      return "Greenhouse water is off";
    }
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Night";
    }
    public String description() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Day";
    }
    public String description() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event {
    public Bell(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Ring every 2 seconds, 'rings' times:
      System.out.println("Bing!");
      if(--rings > 0)
        addEvent(new Bell(
          System.currentTimeMillis() + 200));
    }
    public String description() {
      return "Ring bell";
    }
  }
  class GHEventFactory {
    ArrayList events = new ArrayList();
    class EventCreator {
      Constructor ctor;
      Integer offset;
      public EventCreator(
        Constructor ctor, Integer offset) {
        this.ctor = ctor;
        this.offset = offset;
      }
    }
    public GHEventFactory(String eventFile) {
      try {
        BufferedReader in =
          new BufferedReader(
            new FileReader(eventFile));
        String s;
        while((s = in.readLine())!= null) {
          int colon = s.indexOf(':');
          // Must use '$' instead of '.' to
          // describe inner classes:
          String type = "GreenhouseControls$" +
            s.substring(0, colon).trim();
          Integer offset = new Integer(
            s.substring(colon + 1).trim());
          // Use Reflection from Chapter 10 to
          // find and call the right constructor:
          Class eventClass = Class.forName("c12." + type);
          // Inner class constructors implicitly
          // take the outer-class object as a
          // first argument:
          Constructor ctor = eventClass.getConstructor(
            new Class[]
              { GreenhouseControls.class, long.class });
          events.add(new EventCreator(ctor, offset));
        }
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
    Iterator iterator() {
      return new Iterator() {
        Iterator it = events.iterator();
        public boolean hasNext() {
          return it.hasNext();
        }
        public Object next() {
          long tm = System.currentTimeMillis();
          EventCreator ec = (EventCreator)it.next();
          Object returnVal = null;
          try {
            returnVal = ec.ctor.newInstance(
              new Object[]{
                GreenhouseControls.this,
                new Long(tm + ec.offset.longValue())
              });
          } catch(Exception e) {
            throw new RuntimeException(e);
          }
          return returnVal;
        }
        public void remove() {
          // Do nothing
        }
      };
    }
  }
  static int iterations = 3;
  GHEventFactory gheFactory;
  class Restart extends Event {
    public Restart(long eventTime) {
      super(eventTime);
    }
    public Restart() {
      super(System.currentTimeMillis());
    }
    public void action() {
      if(--iterations <= 0) System.exit(0);
      long tm = System.currentTimeMillis();
      rings = 5;
      Iterator it = gheFactory.iterator();
      while(it.hasNext())
        addEvent((Event)it.next());
    }
    public String description() {
      return "Restarting system";
    }
  }
  public GreenhouseControls(String initFile) {
    gheFactory = new GHEventFactory(initFile);
  }
}
```

```
public class E17_GreenhouseConfigFile {
  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls("E17_GreenhouseConfig.txt");
    gc.addEvent(gc.new Restart());
    gc.run();
  }
} ///:~
```

The idea is to open the external configuration file, and read in the names of the classes and the time offset for each one. The configuration file consists of lines containing the class name, a colon, and the time offset. Here is the configuration file derived from the example in the book.

```
//:! c12:E17_GreenhouseConfig.txt
ThermostatNight: 0
LightOn: 100
LightOff: 200
WaterOn: 300
WaterOff: 800
Bell: 900
ThermostatDay: 1000
Restart: 2000
///:~
```

(Note that the extraction program for the code will remove the first and last lines because of the '**//:!**' on the first line, so those lines do not appear in the file that comes in the code distribution for this book). Note that I've shortened the times so that the example can run in a reasonable time during a build/test of the examples in this book.

The **GHEventFactory** class contains an **ArrayList** to hold an **EventCreator** object for each line in the configuration file. Each **EventCreator** is just a holder for a **Constructor** and the **offset** value for each event. The **Constructor** is a **java.lang.reflection** object that represents a constructor for a class, and once you have such an object you can use it to dynamically create new objects. The trick, as you shall see, is procuring the **Constructor** object, which we see in the **GHEventFactory** constructor.

The **GHEventFactory** constructor takes an argument which is the text configuration file. It opens this, reads each line in, finds the colon and uses that to parse the line into the class name and offset value. Note that since these are inner classes, **GreenhouseControls** must be prepended in order to qualify the name, but it is not separated with a '**.**'! The **Class.forName( )** method is actually looking for the filename to load, so you must separate the names with the '**$**', which is the actual character used to separate the class names in an inner class.

Class.forName( ) is handed a string, and it uses this to produce a reference to the **Class** object. The method **getConstructor( )** is used to produce the **Constructor** object, but it must be given the argument list to match with the appropriate constructor. To do this, you give it an array of **Class** objects matching that argument list, and this array is dynamically created using **new**. At this point, the **EventCreator** can be added.

A number of exceptions could be thrown at this point, but I chose to convert them all into **RuntimeException**s. If anything fails, it reports everything to the console.

I created an iterator to move through the **EventCreator** objects and produce new **Event** objects; this was defined as an anonymous inner class that is built on top of the **events ArrayList**. Every time you call **next( )**, the iterator fetches the next **EventCreator** in the list and uses its **Constructor** and **offset** to build a new **Event** object. The **newInstance** method is called on the **Constructor** object, and it requires the correct number and type of arguments passed to it as a dynamically created array of **Object**s. Note that the offset is added to the current time each time you call **next( )**.

The **Restart** class uses the **GHEventFactory** iterator to move through the list and populate the event manager with **Event** objects.

In **Restart.action( )** I've added a test to limit the number of repetitions so that the program completes in a short time.

Exercise 18

```
//: c12:E18_PrintCharBuffer.java
```

```
/******************** Exercise 18 *********************
 * Create and test a utility method to print the contents
 * of a CharBuffer up to the point where the characters
 * are no longer printable.
 ******************************************************/
import java.nio.*;
import java.util.*;

public class E18_PrintCharBuffer {
  static BitSet isPrintable = new BitSet(127);
  static String encoding =
    System.getProperty("file.encoding");
  static {
    // Assume an encoding that obeys ASCII eg.ISO-8859-1.
    // Characters 32 to 127 represent printable characters.
    for(int i = 32; i <= 127; i++)
      isPrintable.set(i);
  }
  // Set the position to the last printable character
  public static void setPrintableLimit(CharBuffer cb) {
    cb.rewind();
    while(isPrintable.get(cb.get()));
    cb.limit(cb.position() - 1);
    cb.rewind();
  }
  public static void main(String[] args) {
    System.out.println("Default Encoding is: " + encoding);
    CharBuffer buff =
      ByteBuffer.allocate(16).asCharBuffer();
    buff.put("ABCDE" + (char) 0x01 + "FG");
    buff.rewind();
    System.out.println(buff); // Print everything
    setPrintableLimit(buff);
    System.out.println(buff); // Print printable
  }
} ///:~
```

Exercise 19

```
//: c12:E19_AllocateDirect.java
```

```
// {Clean: temp.txt, data2.txt}
/******************** Exercise 19 ************************
 * Experiment with changing the ByteBuffer.allocate()
 * statements in the examples in this chapter to
 * ByteBuffer.allocateDirect(). Demonstrate performance
 * differences, but also notice whether the startup time
 * of the programs noticeably changes.
 *********************************************************/
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import java.nio.charset.*;

abstract class CompareAllocations {
  private String name;
  protected ByteBuffer buff;
  private int size;
  public CompareAllocations(String name, int size) {
    this.name = name;
    this.size = size;
  }
  public void runComparison() {
    System.out.println("Program Name: <" + name + ">");
    try {
      long startTime = System.currentTimeMillis();
      directAllocate();
      long endTime = System.currentTimeMillis();
      System.out.println(
        "Direct Allocation Cost for buffer of size: <"
        + size + "> is <" + (endTime - startTime) + ">");
      startTime = System.currentTimeMillis();
      execute();
      endTime = System.currentTimeMillis();
      System.out.println(
        "Execution cost using direct buffer: <"
          + (endTime - startTime) + ">");
      startTime = System.currentTimeMillis();
      indirectAllocate();
      endTime = System.currentTimeMillis();
      System.out.println(
        "Indirect Allocation Cost for buffer of size: <"
```

```
          + size + "> is <" + (endTime - startTime) + ">");
      startTime = System.currentTimeMillis();
      execute();
      endTime = System.currentTimeMillis();
      System.out.println(
        "Execution cost using indirect buffer: <"
          + (endTime - startTime) + ">");
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void directAllocate() {
    buff = ByteBuffer.allocateDirect(size);
  }
  public abstract void execute() throws IOException;
  public void indirectAllocate() {
    buff = ByteBuffer.allocate(size);
  }
}

public class E19_AllocateDirect {
  public static void main(String[] args) {
    CompareAllocations[] comparisons = {
      new CompareAllocations("GetChannel", 8192) {
        public void execute() throws IOException {
          FileChannel fc =
            new FileInputStream("E19_AllocateDirect.java")
            .getChannel();
          fc.read(buff);
          buff.flip();
          while(buff.hasRemaining())
            buff.get();
        }
      },
      new CompareAllocations("ChannelCopy", 8192) {
        public void execute() throws IOException {
          FileChannel in =
            new FileInputStream("E19_AllocateDirect.java")
            .getChannel(),
            out = new FileOutputStream("temp.txt")
            .getChannel();
          while(in.read(buff) != -1) {
            buff.flip(); // Prepare for writing
            out.write(buff);
            buff.clear();  // Prepare for reading
          }
        }
      },
      new CompareAllocations("BufferToText", 8192) {
        public void execute() throws IOException {
          FileChannel fc =
              new FileOutputStream("data2.txt")
              .getChannel();
         fc.write(ByteBuffer.wrap("Some text".getBytes()));
          fc.close();
          fc = new FileInputStream("data2.txt")
               .getChannel();
          fc.read(buff);
          buff.flip();
          buff.asCharBuffer().toString();
          // Decode using this system's default Charset:
          buff.rewind();
          Charset.forName(
            System.getProperty("file.encoding"))
            .decode(buff);
          fc = new FileOutputStream("data2.txt")
               .getChannel();
          fc.write(ByteBuffer.wrap(
            "Some text".getBytes("UTF-16BE")));
          fc.close();
          // Now try reading again:
          fc = new FileInputStream("data2.txt")
               .getChannel();
          buff.clear();
          fc.read(buff);
          buff.flip();
          buff.asCharBuffer().toString();
          // Use a CharBuffer to write through:
          fc = new FileOutputStream("data2.txt")
               .getChannel();
          buff.clear();
          buff.asCharBuffer().put("Some text");
```

```
          fc.write(buff);
          fc.close();
          // Read and display:
          fc = new FileInputStream("data2.txt")
               .getChannel();
          buff.clear();
          fc.read(buff);
          buff.flip();
          buff.asCharBuffer().toString();
        }
      },
      new CompareAllocations("GetData", 1024) {
        public void execute() throws IOException {
          // Store and read a char array:
          buff.asCharBuffer().put("Howdy!");
          // Store and read a short:
          buff.asShortBuffer().put((short)471142);
          buff.getShort();
          buff.rewind();
          // Store and read an int:
          buff.asIntBuffer().put(99471142);
          buff.getInt();
          buff.rewind();
          // Store and read a long:
          buff.asLongBuffer().put(99471142);
          buff.getLong();
          buff.rewind();
          // Store and read a float:
          buff.asFloatBuffer().put(99471142);
          buff.getFloat();
          buff.rewind();
          // Store and read a double:
          buff.asDoubleBuffer().put(99471142);
          buff.getDouble();
          buff.rewind();
        }
      },
      new CompareAllocations("IntBufferDemo", 1024) {
        public void execute() throws IOException {
          IntBuffer ib = buff.asIntBuffer();
          // Store an array of int:
```

```
          ib.put(
            new int[] { 11, 42, 47, 99, 143, 811, 1016 });
          // Absolute location read and write:
          ib.get(3);
          ib.put(3, 1811);
          ib.rewind();
          while(ib.hasRemaining()) {
            int i = ib.get();
            if(i == 0) break;
          }
        }
      },
      new CompareAllocations("UsingBuffers", 32) {
        public void execute() throws IOException {
          char[] data = "UsingBuffers".toCharArray();
          CharBuffer cb = buff.asCharBuffer();
          cb.put(data);
          cb.rewind();
          symmetricScramble(cb);
          cb.rewind();
          symmetricScramble(cb);
          cb.rewind();
        }
       private void symmetricScramble(CharBuffer buffer) {
          while(buffer.hasRemaining()) {
            buffer.mark();
            char c1 = buffer.get();
            char c2 = buffer.get();
            buffer.reset();
            buffer.put(c2).put(c1);
          }
        }
      }
    };
    for(int i = 0; i < comparisons.length; i++)
      comparisons[i].runComparison();
  }
} ///:~
```

Exercise 20

```
//: c12:E20_CheckForMatch.java
/********************** Exercise 20 *********************
 * For the phrase "Java now has regular expressions"
 * evaluate whether the following expressions will find a
 * match:
 *
 *   ^Java
 *   \Breg.*
 *   n.w\s+h(a|i)s
 *   s?
 *   s*
 *   s+
 *   s{4}
 *   s{1}.
 *   s{0,3}
 *********************************************************/
import java.util.regex.*;

public class E20_CheckForMatch {
  public static void main(String[] args) {
    String source = "Java now has regular expressions";
    String[] regEx = {"^Java", "\\Breg.*",
      "n.w\\s+h(a|i)s", "s?", "s*", "s+", "s{4}", "s{1}.",
      "s{0,3}"};
    System.out.println("Source string: " + source);
    for(int i = 0; i < regEx.length; i++) {
      System.out.println(
        "Regular expression: \"" + regEx[i] + "\"");
      Pattern p = Pattern.compile(regEx[i]);
      Matcher m = p.matcher(source);
      while(m.find()) {
        System.out.println("Match \"" + m.group() +
        "\" at positions " + m.start() + "-" +
        (m.end() - 1));
      }
    }
  }
} ///:~
```

The output is:

```
Source string: Java now has regular expressions
Regular expression: "^Java"
Match "Java" at positions 0-3
Regular expression: "\Breg.*"
Regular expression: "n.w\s+h(a|i)s"
Match "now has" at positions 5-11
Regular expression: "s?"
Match "" at positions 0--1
Match "" at positions 1-0
Match "" at positions 2-1
Match "" at positions 3-2
Match "" at positions 4-3
Match "" at positions 5-4
Match "" at positions 6-5
Match "" at positions 7-6
Match "" at positions 8-7
Match "" at positions 9-8
Match "" at positions 10-9
Match "s" at positions 11-11
Match "" at positions 12-11
Match "" at positions 13-12
Match "" at positions 14-13
Match "" at positions 15-14
Match "" at positions 16-15
Match "" at positions 17-16
Match "" at positions 18-17
Match "" at positions 19-18
Match "" at positions 20-19
Match "" at positions 21-20
Match "" at positions 22-21
Match "" at positions 23-22
Match "" at positions 24-23
Match "" at positions 25-24
Match "s" at positions 26-26
Match "s" at positions 27-27
Match "" at positions 28-27
Match "" at positions 29-28
Match "" at positions 30-29
Match "s" at positions 31-31
Match "" at positions 32-31
Regular expression: "s*"
Match "" at positions 0--1
Match "" at positions 1-0
Match "" at positions 2-1
Match "" at positions 3-2
Match "" at positions 4-3
Match "" at positions 5-4
Match "" at positions 6-5
Match "" at positions 7-6
Match "" at positions 8-7
Match "" at positions 9-8
Match "" at positions 10-9
Match "s" at positions 11-11
Match "" at positions 12-11
Match "" at positions 13-12
Match "" at positions 14-13
Match "" at positions 15-14
Match "" at positions 16-15
Match "" at positions 17-16
Match "" at positions 18-17
Match "" at positions 19-18
Match "" at positions 20-19
Match "" at positions 21-20
Match "" at positions 22-21
Match "" at positions 23-22
Match "" at positions 24-23
Match "" at positions 25-24
Match "ss" at positions 26-27
Match "" at positions 28-27
Match "" at positions 29-28
Match "" at positions 30-29
Match "s" at positions 31-31
Match "" at positions 32-31
Regular expression: "s+"
Match "s" at positions 11-11
Match "ss" at positions 26-27
Match "s" at positions 31-31
Regular expression: "s{4}"
Regular expression: "s{1}."
Match "s " at positions 11-12
Match "ss" at positions 26-27
Regular expression: "s{0,3}"
Match "" at positions 0--1
Match "" at positions 1-0
Match "" at positions 2-1
Match "" at positions 3-2
Match "" at positions 4-3
Match "" at positions 5-4
Match "" at positions 6-5
Match "" at positions 7-6
Match "" at positions 8-7
Match "" at positions 9-8
Match "" at positions 10-9
Match "s" at positions 11-11
Match "" at positions 12-11
Match "" at positions 13-12
Match "" at positions 14-13
Match "" at positions 15-14
Match "" at positions 16-15
Match "" at positions 17-16
Match "" at positions 18-17
Match "" at positions 19-18
Match "" at positions 20-19
Match "" at positions 21-20
Match "" at positions 22-21
Match "" at positions 23-22
Match "" at positions 24-23
Match "" at positions 25-24
Match "ss" at positions 26-27
Match "" at positions 28-27
Match "" at positions 29-28
Match "" at positions 30-29
Match "s" at positions 31-31
Match "" at positions 32-31
```

Exercise 21

```
//: c12:E21_CheckForMatch2.java
/************************* Exercise 21 *****************
 *Apply the regular expression
 *      (?i)((^[aeiou])|(\s+[aeiou]))\w+?[aeiou]\b
 *     to
 *    "Arline ate eight apples and one orange while Anita
 *     hadn't any"
 *********************************************************/
import java.util.regex.*;

public class E21_CheckForMatch2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(
      "(?i)((^[aeiou])|(\\s+[aeiou]))\\w+?[aeiou]\\b");
    Matcher m = p.matcher("Arline ate eight apples and " +
      "one orange while Anita hadn't any");
    while(m.find()) {
      System.out.println("Match \"" + m.group() +
        "\" at positions " + m.start() + "-" +
        (m.end() - 1));
    }
  }
} ///:~
```

The output is:

```
Match "Arline" at positions 0-5
Match " ate" at positions 6-9
Match " one" at positions 27-30
Match " orange" at positions 31-37
Match " Anita" at positions 44-49
```

Exercise 22

```
//: c12:E22_JGrep2.java
// {Args: E22_JGrep2.java "\\b[Ssct]\\w+" CASE_INSENSITIVE}
/********************** Exercise 22 ********************
 * Modify JGrep.java to accept flags as arguments (e.g.,
 * Pattern.CASE_INSENSITIVE, Pattern.MULTILINE).
 *********************************************************/
import java.util.regex.*;
import java.util.*;
import com.bruceeckel.util.*;
```

```
public class E22_JGrep2 {
  public static void main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println(
        "Usage: java JGrep file regex pattern");
      System.out.println(
        "pattern can take one of the following values");
      System.out.println(
        "CANON_EQ, CASE_INSENSITIVE, COMMENTS, DOTALL, " +
        "MULTILINE, UNICODE_CASE, UNIX_LINES");
      System.exit(0);
    }
    int flag = 0;
    if(args[2].equalsIgnoreCase("CANON_EQ")) {
      flag = Pattern.CANON_EQ;
    } else
      if(args[2].equalsIgnoreCase("CASE_INSENSITIVE")) {
      flag = Pattern.CASE_INSENSITIVE;
    } else if(args[2].equalsIgnoreCase("COMMENTS")) {
      flag = Pattern.COMMENTS;
    } else if(args[2].equalsIgnoreCase("DOTALL")) {
      flag = Pattern.DOTALL;
    } else if(args[2].equalsIgnoreCase("MULTILINE")) {
      flag = Pattern.MULTILINE;
    } else if(args[2].equalsIgnoreCase("UNICODE_CASE")) {
      flag = Pattern.UNICODE_CASE;
    } else if(args[2].equalsIgnoreCase("UNIX_LINES")) {
      flag = Pattern.UNIX_LINES;
    }
    Pattern p = Pattern.compile(args[1], flag);
    // Iterate through the lines of the input file:
    ListIterator it = new TextFile(args[0]).listIterator();
    while(it.hasNext()) {
      Matcher m = p.matcher((String)it.next());
      while(m.find())
        System.out.println(it.nextIndex() + ": " +
          m.group() + ": " + m.start());
    }
  }
} ///:~
```

The output is:

```
1: c12: 4
2: c12: 8
3: Ssct: 31
3: CASE_INSENSITIVE: 42
5: to: 21
6: CASE_INSENSITIVE: 11
10: com: 7
12: class: 7
13: static: 9
13: String: 26
13: throws: 41
15: System: 6
17: System: 6
18: can: 17
18: take: 21
18: the: 33
19: System: 6
20: CANON_EQ: 9
20: CASE_INSENSITIVE: 19
20: COMMENTS: 37
22: System: 6
25: CANON_EQ: 33
26: CANON_EQ: 21
27: CASE_INSENSITIVE: 40
28: CASE_INSENSITIVE: 21
29: COMMENTS: 40
30: COMMENTS: 21
40: compile: 24
41: through: 15
41: the: 23
41: the: 36
42: TextFile: 26
44: String: 29
46: System: 8
47: start: 31
```

Exercise 23

```
//: c12:E23_JGrep3.java
// {Args: E23_JGrep3.java "\\b[Ssct]\\w+"}
/********************** Exercise 23 *********************
 * Modify JGrep.java to use Java nio memory-mapped files.
 *********************************************************/
import java.io.*;
import java.util.regex.*;
import java.nio.channels.*;
import java.nio.*;
import java.nio.charset.*;

public class E23_JGrep3 {
  public static void main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println("Usage: java JGrep file regex");
      System.exit(0);
    }
    Pattern p = Pattern.compile(args[1]);
    FileChannel fc =
      new FileInputStream(args[0]).getChannel();
    ByteBuffer buff =
      fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());
    CharBuffer cb = Charset.forName(
      System.getProperty("file.encoding")).decode(buff);
    String[] fileAsArray =
      Pattern.compile("\n").split(cb.toString());
    // You can squeeze the above two statements in one
    // String[] fileAsArray =
    //  Pattern.compile("\n").split(Charset.forName(
    //  System.getProperty("file.encoding")).decode(buff));
    for(int i = 0; i < fileAsArray.length; i++) {
      Matcher m = p.matcher(fileAsArray[i]);
      while(m.find())
        System.out.println((i + 1) + ": " + m.group() +
          ": " + m.start());
    }
    fc.close();
  }
} ///:~
```

Exercise 24

```
//: c12:E24_JGrep4.java
// {Args: E24_JGrep4.java java }
/********************** Exercise 24 ********************
 * Modify JGrep.java to accept a directory name or a file
 * name as argument (if a directory is provided, search
 * should include all files in the directory).
 * Hint: you can generate a list of filenames with
 * String[] filenames = new File(".").list();
 ******************************************************/
import java.io.*;
import java.util.regex.*;
import java.nio.channels.*;
import java.nio.*;
import java.nio.charset.*;

public class E24_JGrep4 {
  public static void main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println(
        "Usage: java E24_JGrep4 file regex");
      System.exit(0);
    }
    Pattern p = Pattern.compile(args[1]);
    File file = new File(args[0]);
    String[] fileName = null;
    if(file.isDirectory()) fileName = file.list();
    else fileName = new String[]{args[0]};
    for(int i = 0; i < fileName.length; i++) {
      System.out.println("-- File:" + fileName[i] + "--");
      // Cannot do a fc.read() on a directory.
      if(new File(fileName[i]).isDirectory()) continue;
      FileChannel fc =
        new FileInputStream(fileName[i]).getChannel();
      ByteBuffer buff =
        ByteBuffer.allocate((int)fc.size());
      fc.read(buff);
      buff.flip();
      CharBuffer cb = Charset.forName(
        System.getProperty("file.encoding")).decode(buff);
      String[] fileAsArray =
        Pattern.compile("\n").split(cb.toString());
      for(int j = 0; j < fileAsArray.length; j++) {
        Matcher m = p.matcher(fileAsArray[j]);
        while(m.find())
          System.out.println((j + 1) + ": " + m.group() +
            ": " + m.start());
      }
      fc.close();
    }
  }
} ///:~
```

Chapter 13

Exercise 1

```
//: c13:E01_BasicThreading.java
/****************** Exercise 1 ******************
 * Inherit a class from Thread and override the
 * run() method. Inside run(), print a message,
 * and then call sleep(). Repeat this three
 * times, then return from run(). Put a start-up
 * message in the constructor and override
 * finalize() to print a shut-down message. Make
 * a separate thread class that calls System.gc()
 * and System.runFinalization() inside run(),
 * printing a message as it does so. Make several
 * thread objects of both types and run them to
 * see what happens.
 ***********************************************/
class Cleaner extends Thread { // The "separate" class.
  static int counter = 0;
  int id = counter++;
  public Cleaner() {
    start();
  }
  public void run() {
    try {
      sleep(3000);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println("Cleaner " + id + " run");
    System.gc();
    System.runFinalization();
  }
}

public class E01_BasicThreading extends Thread {
  static int counter = 0;
  int id = counter++;
  public E01_BasicThreading() {
    System.out.println("Constructing thread "+id);
    start();
  }
  public void run() {
    for(int i = 0; i < 3; i++) {
      System.out.println("Basic Thread " + id
        + " Loop " + i);
      try {
        sleep(2000);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
  protected void finalize() {
    System.out.println("Basic Thread " + id + " Finalize");
  }
  public static void
  main(String args[]) throws InterruptedException {
    for(int i = 0; i < 5; i++) {
      new E01_BasicThreading();
      new Cleaner();
      sleep(1000);
    }
  }
} ///:~
```

The output for one run is:

```
Constructing thread 0
Basic Thread 0 Loop 0
Constructing thread 1
Basic Thread 1 Loop 0
Constructing thread 2
Basic Thread 0 Loop 1
Basic Thread 2 Loop 0
Cleaner 0 run
Basic Thread 1 Loop 1
Constructing thread 3
```

```
Basic Thread 3 Loop 0
Basic Thread 0 Loop 2
Cleaner 1 run
Basic Thread 2 Loop 1
Constructing thread 4
Basic Thread 4 Loop 0
Cleaner 2 run
Basic Thread 1 Loop 2
Basic Thread 3 Loop 1
Basic Thread 2 Loop 2
Cleaner 3 run
Basic Thread 4 Loop 1
Basic Thread 0 Finalize
Basic Thread 3 Loop 2
Cleaner 4 run
Basic Thread 1 Finalize
Basic Thread 4 Loop 2
```

Notice that not all the finalizers get called. Try playing with the **sleep( )** delays to see what effect it has.

Exercise 2

```
//: c13:E02_Daemons2.java
// {Args: 5}
/****************** Exercise 2 ***********************
 * Experiment with different sleep times in Daemons.java
 * to see what happens.
 ******************************************************/
class Daemon extends Thread {
  private Thread[] t = new Thread[10];
  public Daemon() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < t.length; i++)
      t[i] = new DaemonSpawn(i);
    for(int i = 0; i < t.length; i++)
      System.out.println("t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true)
      yield();
  }
}

class DaemonSpawn extends Thread {
  public DaemonSpawn(int i) {
    start();
    System.out.println("DaemonSpawn " + i + " started");
  }
  public void run() {
    while(true)
      yield();
  }
}

public class E02_Daemons2 {
  public static void main(String[] args) throws Exception {
    if(args.length < 1) {
      System.out.println(
        "Usage: java E02_Daemons2 <sleep time>");
      System.exit(1);
    }
    Thread d = new Daemon();
    System.out.println("d.isDaemon() = " + d.isDaemon());
    Thread.sleep(Integer.parseInt(args[0]));
  }
} ///:~
```

Exercise 3

```
//: c13:E03_GreenhouseThread.java
/****************** Exercise 3 ******************
 * In Chapter 8, locate the
 * GreenhouseController.java example, which
 * consists of four files. In Event.java, the
 * class Event is based on watching the time.
 * Change Event so that it is a Thread, and
 * change the rest of the design so that it
 * works with this new Thread-based Event.
 **********************************************/
abstract class Event extends Thread {
  private long delay;
  public Event(long delayTime) {
    delay = delayTime;
    start();
  }
  public void run() {
    try {
      sleep(delay);
    } catch(InterruptedException e) {
      System.out.println("InterruptedException = " + e);
    }
    action();
    System.out.println(this);
  }
  abstract public void action();
}

class GreenhouseControls {
//  List events = new ArrayList();
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String toString() {
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String toString() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here
      water = true;
    }
    public String toString() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here
      water = false;
    }
    public String toString() {
      return "Greenhouse water is off";
    }
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Night";
    }
    public String toString() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Day";
    }
    public String toString() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event {
    public Bell(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Ring every 2 seconds, 'rings' times:
      System.out.println("Bing!");
      if(--rings > 0)
        new Bell(200);
    }
    public String toString() {
      return "Ring bell";
    }
  }
  static int repeats = 3;
  class Restart extends Event {
    public Restart(long delayTime) {
      super(delayTime);
    }
    public void action() {
      if(--repeats < 0) return;
      rings = 5;
      new ThermostatNight(0);
      new LightOn(100);
      new LightOff(200);
      new WaterOn(300);
      new WaterOff(800);
      new Bell(900);
      new ThermostatDay(1000);
      // Can even add a Restart object!
      new Restart(2000);
    }
    public String toString() {
      return "Restarting system";
    }
  }
}

public class E03_GreenhouseThread {
  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    gc.new Restart(1000);
  }
} ///:~
```

By using threading the code is dramatically simplified. The **EventSet** and **Controller** classes have been completely eliminated. Most of the code in **GreenhouseControls** remains the same, but the **Event** class is quite different – it is a **Thread**, and its **run( )** method just sleeps for the desired delay time, then runs **action( )** and prints **description( )** and then returns – so not only do you not have to worry about removing the thread from an event queue, you don't need an event queue at all! Just creating an **Event** object ensures that it will be run, with no need for a controller to be constantly checking a list of events for ones ready to be run. The thread mechanism takes care of all of this for you.

One other change: when you create an **Event**, you just give the delay to run, rather than the absolute time.

Exercise 4

```
//: c13:E04_GreenHouseTimer.java
/****************** Exercise 4 ******************
 * Modify the previous exercise so that the
 * java.util.Timer class is used to run the
 * system.
 ***********************************************/
import java.util.*;

abstract class Event_T extends TimerTask {
  private long delayTime;
  public Event_T(long eventDelay) {
    delayTime = eventDelay;
   // Causes NullPointerException since a TimerTask may
   // be scheduled before it is completely construted.
   // new Timer().schedule(this, delayTime);
  }
  public void schedule() {
    new Timer().schedule(this, delayTime);
  }
  // abstract public void action();
  // Replace "action" with TimerTask's run()::
  public void run() {
    System.out.println(this);
  }
}

class GreenhouseControls_T {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event_T {
    public LightOn(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
      super.run();
    }
    public String toString() { return "Light is on"; }
  }
  private class LightOff extends Event_T {
    public LightOff(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
      super.run();
    }
    public String toString() { return "Light is off"; }
  }
  private class WaterOn extends Event_T {
    public WaterOn(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Put hardware control code here
      water = true;
      super.run();
    }
    public String toString() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event_T {
    public WaterOff(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Put hardware control code here
      water = false;
      super.run();
    }
    public String toString() {
      return "Greenhouse water is off";
    }
  }
  private class ThermostatNight extends Event_T {
    public ThermostatNight(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Put hardware control code here
      thermostat = "Night";
      super.run();
    }
    public String toString() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event_T {
    public ThermostatDay(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Put hardware control code here
      thermostat = "Day";
      super.run();
    }
    public String toString() {
      return "Thermostat on day setting";
    }
  }
  // An example of an run() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event_T {
    public Bell(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      // Ring every 2 seconds, 'rings' times:
      System.out.println("Bing!");
      if(--rings > 0)
        new Bell(200).schedule();
      super.run();
    }
    public String toString() { return "Ring bell"; }
  }
  static int repeats = 3;
  class Restart extends Event_T {
    public Restart(long eventDelay) {
      super(eventDelay);
    }
    public void run() {
      super.run();
      if(--repeats < 0) {
        System.out.println("Finished");
        // Must call this or threads keep the
        // program running:
        System.exit(0);
      }
      rings = 5;
      new ThermostatNight(0).schedule();
      new LightOn(100).schedule();
      new LightOff(200).schedule();
      new WaterOn(300).schedule();
      new WaterOff(800).schedule();
      new Bell(900).schedule();
      new ThermostatDay(1000).schedule();
      // Can even add a Restart object!
      new Restart(2000).schedule();
    }
    public String toString() {
      return "Restarting system";
    }
  }
}

public class E04_GreenHouseTimer {
  public static void main(String[] args) {
    GreenhouseControls_T gc =
      new GreenhouseControls_T();
    gc.new Restart(0).schedule();
  }
} ///:~
```

This has the same basic design and many of the same benefits as the previous solution, except that the **Timer** class is more flexible in the various ways that you can schedule timers. In addition, you must explicitly call **System.exit(0)** to finish the program, since otherwise the threads keep the program running.

Exercise 5

```
//: c13:E05_SimpleThreadDaemon.java
/****************** Exercise 5 ************************
 * Modify SimpleThread.java so that all the threads
 * are daemon threads, and verify that the program ends
 * as soon as main() is able to exit.
 *****************************************************/
public class E05_SimpleThreadDaemon extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  public E05_SimpleThreadDaemon() {
    super("" + ++threadCount); // Store the thread name
    setDaemon(true);
    start();
  }
  public String toString() {
    return "#" + getName() + ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      try {
        sleep(100);  // Introduce delay.
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new E05_SimpleThreadDaemon();
  }
} ///:~
```

When you run the program you'll see that the daemon threads are unable to complete their countdowns before the program terminates.

Exercise 6

```
//: c13:E06_ManyTimers.java
// {Args: 100}
/****************** Exercise 6 ************************
 * Demonstrate that java.util.Timer scales to large
 * numbers by creating a program that generates many
 * Timer objects that perform some simple task when
 * the timeout completes (if you want to get fancy,
 * you can jump forward to the "Windows and Applets"
 * chapter and use the Timer objects to draw pixels on
 * the screen, but printing to the console is
 * sufficient).
 ******************************************************/
import java.util.*;

public class E06_ManyTimers {
  public static void main(String[] args) throws Exception {
    if(args.length < 1) {
      System.out.println(
        "Usage: java E06_ManyTimers <num of timers>");
    }
    int numOfTimers = Integer.parseInt(args[0]);
    for(int i = 0; i < numOfTimers; i++) {
      new Timer().schedule(new TimerTask() {
        public void run() {
          System.out.println(System.currentTimeMillis());
        }
      }, numOfTimers - i);
    }
    // Wait for timers to expire
    Thread.sleep(2 * numOfTimers);
    System.exit(0);
  }
} ///:~
```

Exercise 7

```
//: c13:E07_SyncChain.java
/****************** Exercise 7 ***********************
 * Demonstrate that a synchronized method in a class
 * can call a second synchronized method in the same
 * class, which can then call a third synchronized
 * method in the same class. Create a separate Thread
 * object that invokes the first synchronized method.
 ******************************************************/
public class E07_SyncChain {
  private static synchronized void m1() {
    System.out.println("In m1()");
    m2();
  }
  private static synchronized void m2() {
    System.out.println("In m2()");
    m3();
  }
  private static synchronized void m3() {
    System.out.println("In m3()");
  }
  public static void main(String[] args) {
    new Thread() {
      public void run() { m1(); }
    }.start();
  }
} ///:~
```

The output is:

```
In m1()
In m2()
In m3()
```

If one **synchronized** method couldn't call another one in the same object, the program would hang after calling **m1()**.

Exercise 8

```
//: c13:E08_ThreadCooperation.java
/****************** Exercise 8 ******************
 * Create two Thread subclasses, one with a run()
 * that starts up and then calls wait(). The
 * other class should capture the reference of
 * the first Thread object. Its run() should call
 * notifyAll() for the first thread after some
 * number of seconds have passed, so the first
 * thread can print a message.
 ***********************************************/
class Coop1 extends Thread {
  public Coop1() {
    System.out.println("Constructed Coop1");
    start();
  }
  public void run() {
    System.out.println("Coop1 going into wait");
    synchronized(this) {
      try {
        wait();
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
    System.out.println("Coop1 exited wait");
  }
}

class Coop2 extends Thread {
  Coop1 otherThread;
  public Coop2(Coop1 otherThread) {
    this.otherThread = otherThread;
    System.out.println("Constructed Coop2");
    start();
  }
  public void run() {
    System.out.println("Coop2 pausing 5 secs");
    try {
      sleep(5000);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println("Coop2 calling notifyAll");
    synchronized(otherThread) {
      otherThread.notifyAll();
    }
  }
}

public class E08_ThreadCooperation {
  public static void main(String args[]) {
    new Coop2(new Coop1());
  }
} ///:~
```

Another example of a badly-worded exercise, which I've reworded to correctly express my intent (hard to find these things unless you actually *work* the exercise ☺).

The output is:

```
Constructed Coop1
Constructed Coop2
Coop2 pausing 5 secs
Coop1 going into wait
Coop2 calling notifyAll
Coop1 exited wait
```

Additional Exercise

Create three classes. The first class, **Store**, should store a number, the second class should create a **Thread** that starts counting (0, 1, 2, 3, etc.) and stores each number in the **Store** class. The third class should create a **Thread** that keeps reading each value in the **Store** class and printing it. The goal is to use synchronization and the **wait( )** and **notify( )** methods to communicate between the **Thread**s so that each number is printed once and only once.

Solution (contributed from Sweden):

```
//: c13:E08B_ThreadCommunication.java

class Store {
  private int value;
  private static volatile boolean available;
  public synchronized void setValue(int i) {
    waitUntilAvailabilityIs(false);
    value = i;
    setAvailability(true);
  }
  public synchronized int getValue() {
    waitUntilAvailabilityIs(true);
    setAvailability(false);
    return value;
  }
  // Utility methods to keep from duplicating code:
  private void
  waitUntilAvailabilityIs(boolean availability) {
    while(available != availability)
      try {
        wait();
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
  }
  private void setAvailability(boolean availability) {
    available = availability;
    notify();
  }
}

class Counter extends Thread {
  private Store s;
  public Counter(Store store) {
    s = store;
    start();
  }
  public void run() {
    for(int i = 0; i < 10000; i++)
```

```
      s.setValue(i);
    }
  }

class Printer extends Thread {
  private Store s;
  public Printer(Store store) {
    s = store;
    setDaemon(true);
    start();
  }
  public void run() {
    while(true)
      System.out.println(s.getValue());
  }
}

public class E08B_ThreadCommunication {
  public static void main(String[] args) {
    Store store = new Store();
    Counter counter = new Counter(store);
    Printer printer = new Printer(store);
  }
} ///:~
```

The goal is to make it trivial for **Counter** to send information to **Printer** via a **Store** object, and you can see how simple the implementations of **Counter** and **Printer** are, as a result of hiding all the complexity in the **Store** class.

Store holds an **int value**, which can contain any numerical value and thus cannot be used as an indicator of whether it is holding valid information or is "empty." To do that, the **available** flag is used. When **setValue( )** is called, it must first wait until the single **value** space is available before placing a new value in it, or else the previous value would be lost. When **getValue( )** is called, it must wait until a valid value is present before fetching it, otherwise it will produce duplicate results. To accomplish this without duplicating code, the **waitUntilAvailabilityIs( )** method uses **wait( )** to wait until the **available** flag changes to the desired state, and the **setAvailability( )** method changes the **available** flag and calls **notify( )** to stop the

wait( ). Both these methods are **private** because they are only used within **Store**.

Exercise 9

```
//: c13:E09_BusyWait.java
/****************** Exercise 9 ******************
 * Create an example of a "busy wait." One thread sleeps
 * for awhile and then sets a flag to true. The second
 * thread watches that flag inside a while loop (this is
 * the "busy wait") and when the flag becomes true, sets
 * it back to false and reports the change to the
 * console. Note how much wasted time the program
 * spends inside the "busy wait," and create a second
 * version of the program that uses wait() instead of
 * the "busy wait."
 ***********************************************/
public class E09_BusyWait {
  private static volatile boolean flag = false;
  private static int spins = 0;
  public static void main(String[] args) {
    new Timeout(1000, "Timed Out");
    Thread t = new Thread() {
      public void run() {
        while(true) {
          try {
            sleep(10);
          } catch (InterruptedException e) {
            throw new RuntimeException(e);
          }
          flag = true;
        }
      }
    };
    t.start();
    for(;;) {
      while(!flag) // The busy-wait
        spins++;
      System.out.println("Spun " + spins + " times");
      spins = 0;
      flag = false;
    }
  }
} ///:~
```

Here, the communication between the two threads (**t** and the unnamed **main()** thread) happens via **flag**. Normally, you might see a busy-wait as simply

```
while(!flag)
  ;
```

But here we are also keeping track of the amount of activity in the busy-wait loop. If you run the program you'll see that it's called "busy wait" for a good reason.

In contrast, if we use **wait()** and **notify()**, there's no need for a flag because the communication occurs via the threading mechanism.

```
//: c13:E09_WaitNotify.java
// The second version using wait().

public class E09_WaitNotify {
  public static void main(String[] args) throws Exception {
    new Timeout(1000, "Timed Out");
    Thread t = new Thread() {
      public void run() {
        while(true) {
          try {
            sleep(100);
            synchronized(this) {
              notify();
            }
          } catch(InterruptedException e) {
            throw new RuntimeException(e);
          }
        }
      }
    };
    t.start();
    while(true) {
      synchronized(t) {
        t.wait();
      }
      System.out.println("Cycled");
    }
  }
} ///:~
```

The **main()** thread calls **wait()** on **t**, and **t** calls **notify()** on itself.

Exercise 10

```
//: c13:E10_Restaurant2.java
/****************** Exercise 10 ***********************
 * Modify Restaurant.java to use notifyAll() and observe
 * any difference in behavior.
 ***************************************************/

class Order {
  private static int i = 0;
  private int count = i++;
  public Order() {
    if(count == 10) {
      System.out.println("Out of food, closing");
      System.exit(0);
    }
  }
  public String toString() { return "Order " + count; }
}

class WaitPerson extends Thread {
  private E10_Restaurant2 restaurant;
  public WaitPerson(E10_Restaurant2 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(true) {
      while(restaurant.order == null)
        synchronized(this) {
          try {
            wait();
          } catch(InterruptedException e) {
            throw new RuntimeException(e);
          }
        }
      System.out.println(
        "Waitperson got " + restaurant.order);
      restaurant.order = null;
    }
  }
}

class Chef extends Thread {
  private E10_Restaurant2 restaurant;
  private WaitPerson waitPerson;
  public Chef(E10_Restaurant2 r, WaitPerson w) {
    restaurant = r;
    waitPerson = w;
    start();
  }
  public void run() {
    while(true) {
      if(restaurant.order == null) {
        restaurant.order = new Order();
        System.out.print("Order up! ");
        synchronized(waitPerson) {
          waitPerson.notifyAll();
        }
      }
      try {
        sleep(100);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

public class E10_Restaurant2 {
  Order order; // Package access
  public static void main(String[] args) {
    E10_Restaurant2 restaurant = new E10_Restaurant2();
    WaitPerson waitPerson = new WaitPerson(restaurant);
    Chef chef = new Chef(restaurant, waitPerson);
  }
} ///:~
```

There's no difference in behavior because only one thread is waiting, so **notifyAll()** is only waking up one thread.

Exercise 11

```
//: c13:E11_Restaurant3.java
/****************** Exercise 11 ***********************
 * Modify Restaurant.java so that there are multiple
 * WaitPersons, and indicate which one gets each Order.
 ******************************************************/
import java.util.*;

class Order2 {
  private static int i = 0;
  private int count = i++;
  public Order2() {
    if(count == 20) {
      System.out.println("Out of food, closing");
      System.exit(0);
    }
  }
  public String toString() { return "Order " + count; }
}

class WaitPerson2 extends Thread {
  private int id;
  private E11_Restaurant3 restaurant;
  public WaitPerson2(E11_Restaurant3 r, int id) {
    this.id = id;
    restaurant = r;
    start();
  }
  public void run() {
    while(true) {
      while(restaurant.order == null)
        synchronized(restaurant) {
          try {
            // All threads wait on the same object.
            restaurant.wait();
          } catch(InterruptedException e) {
            throw new RuntimeException(e);
          }
        }
      System.out.println(this + " got order " +
        restaurant.order);
      restaurant.order = null;

    }
  }
  public String toString() {
    return "" + id;
  }
}

class Chef2 extends Thread {
  private E11_Restaurant3 restaurant;
  private Random rand = new Random();
  public Chef2(E11_Restaurant3 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(true) {
      if(restaurant.order == null) {
        restaurant.order = new Order2();
        System.out.print("Order up! ");
        synchronized(restaurant) {
          restaurant.notify();
//!       restaurant.notifyAll(); // Doesn't work!
        }
      }
      try {
        sleep(100);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

public class E11_Restaurant3 {
  volatile Order2 order; // Package access
  public static void main(String[] args) {
    E11_Restaurant3 restaurant = new E11_Restaurant3();
    for(int i = 0; i < 5; i++)
      new WaitPerson2(restaurant, i);
    Chef2 chef = new Chef2(restaurant);
  }
} ///:~
```

Because you have multiple threads waiting on a single object, **notifyAll()** is not the call to use in this case, because it would then notify all the **WaitPerson2** objects and they would end up fighting over some of the orders. Here, **notify()** is correct because you only want one **WaitPerson2** to pick up one order. Try changing **notify()** to **notifyAll()** as shown in the code above and you'll see what happens when you run the program.

Exercise 12

```
//: c13:E12_Restaurant4.java
// {Args: 40}
/****************** Exercise 12 ************************
 * Modify Restaurant.java so that multiple WaitPersons
 * generate order requests to multiple Chefs, who produce
 * orders and notify the WaitPerson that generated the
 * request. You'll need to use queues for both incoming
 * order requests and outgoing orders.
*****************************************************/
class Order3 {
  private static int i = 0;
  private int count = i++;
  private WaitPerson3 originator;
  public Order3(WaitPerson3 origin) {
    originator = origin; // Waiter that placed the order
  }
  // Who should deliver this meal?
  public WaitPerson3 destination() { return originator; }
  public String toString() { return "Order " + count; }
  public int number() { return count; }
}

class WaitPerson3 extends Thread {
  private static int i = 0;
  private int count = i++;
  private E12_Restaurant4 restaurant;
  private TQueue meals = new TQueue();
  public WaitPerson3(E12_Restaurant4 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(!isInterrupted()) {
      // Simplest approach -- one order at a time:
      Order3 order = new Order3(this);
      restaurant.putOrder(order);
      // TQueue waits upon a get() until there's a meal:
      order = (Order3)meals.get();
      if(order == null) // get() returned via interrupt
        break; // Out of while loop
      System.out.println(this + " delivers " + order);
    }
    System.out.println(this + " quitting");
  }
  public void putMeal(Order3 order) { meals.put(order); }
  public String toString() { return "WaitPerson " + count;}
}

class Chef3 extends Thread {
  private static int i = 0;
  private int count = i++;
  private E12_Restaurant4 restaurant;
  public Chef3(E12_Restaurant4 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(!isInterrupted()) {
      // TQueue waits until there's an order:
      Order3 order = restaurant.getOrder();
      if(order == null) // get() returned via interrupt
        break; // Out of while loop
      System.out.println(this + " got " + order);
      // Create the meal
      // ...
      // Put the order back on the queue of the waiter
      // that placed it:
      order.destination().putMeal(order);
    }
    System.out.println(this + " quitting");
  }
  public String toString() { return "Chef " + count;}
}

public class E12_Restaurant4 {
  private TQueue orders = new TQueue();
  private Chef3[] chefs = new Chef3[3];
  private WaitPerson3[] waiters = new WaitPerson3[6];
  private int maxOrders;
  public E12_Restaurant4(int maxOrders) {
    this.maxOrders = maxOrders;
    for(int i = 0; i < chefs.length; i++)
      chefs[i] = new Chef3(this);
    for(int i = 0; i < waiters.length; i++)
      waiters[i] = new WaitPerson3(this);
  }
  public synchronized void putOrder(Order3 newOrder) {
    if(newOrder.number() > maxOrders)
      shutdown();
    else
      orders.put(newOrder);
  }
  // No need for extra synchronization on this one;
  // get() already provides synchronization:
  public Order3 getOrder() { return (Order3)orders.get(); }
  public synchronized void shutdown() {
    for(int i = 0; i < chefs.length; i++)
      chefs[i].interrupt();
    for(int i = 0; i < waiters.length; i++)
      waiters[i].interrupt();
  }
  public static void main(String[] args) {
    if(args.length < 1) {
      System.err.println("argument: number of meals");
      System.exit(1);
    }
    new E12_Restaurant4(Integer.parseInt(args[0]));
  }
} ///:~
```

What I've created here is the simplest approach I could think of, but your approach may be simpler. However, the approach I took points out that there's a number of rather important issues missing from the book, and which will need to be included in Thinking in Java, 4th edition.

One feature that's only lightly touched upon is the concept of the **interrupt()**. The example in the book doesn't do it justice – if you have a number of threads that are waiting (or doing something else that blocks them) and you need to shut down the program, they are not going to wake up and look at a flag. Unless, that is, you call **interrupt()** on the waiting threads. We will use **interrupt()** in the solution presented here.

I also didn't present or emphasize the value of queues in order to solve threading problems. In effect, queues provide another kind of serialization mechanism, but in this case they serialize the flow of objects between threads. Queues are important because they provide a single point through which objects can flow, and thus prevent multiple threads from bombarding a recipient with objects.

Here is the queue used in this solution and in the next:

```
//: c13:TQueue.java
// A queue for solving thread synchronization problems.
import java.util.*;

public class TQueue {
  private LinkedList queue = new LinkedList();
  public synchronized void put(Object v) {
    queue.addFirst(v);
    notifyAll();
  }
  public synchronized Object get() {
    while(queue.isEmpty())
      try {
        wait();
      } catch(InterruptedException e) {
        return null; // Says "quit now"
      }
    return queue.removeLast();
  }
} ///:~
```

As shown in Chapter 11 of Thinking in Jave 3rd edition, it is a wrapper around the queue functionality of a **LinkedList**. However, it also involves synchronization code. When a user of a **TQueue** calls **get()** on an empty **TQueue**, that call automatically waits until someone else puts an object into the **TQueue**. Notice the **put()** calls **notifyAll()** so that all threads that are waiting inside a **get()** are woken up. Only the first of these threads will get the available object; the others will see a **true** value for **queue.isEmpty()** and go back to sleep. This provides an excellent way to manage the flow of objects from one thread to another.

A problem occurs if you have multiple threads using multiple **TQueues**. If these multiple threads are calling **get()**, many of them might be in **wait()**s. If you want to shut everything down, you can call **interrupt()** to break a thread out of a **wait()** call. That's what we'll do here, but if an **interrupt()** does occur, we'll indicate it to the caller by returning **null** instead of an object.

Each waiter gets orders and puts them on the queue to be fulfilled. Any chef can fulfill an order, but the order must go back to the waiter who originated it. Thus, the identity of the originating waiter is attached to the order. This mimics the way the paper order would have the waiter's name upon it, so the waiter could be called when it is ready.

Now, if you look back at **E12_Restaurant4.java** you can see that a waiter creates an order, and puts the order in the restaurant's **TQueue**. This will wake up all chefs waiting for orders to fill (note that **TQueue** contains a **while** loop to handle the case when another chef plucks out the order as this chef was waking up from the **wait()**). The chef fills the order, and then that chef must give the order back to the waiter *who originated the order*. How to do this? Well, the order will already need to contain a reference to the waiter that originated it, for one thing. And you could conceivably put the order back on a second queue within the restaurant and notify all the waiters who could show up and look at the order and figure out if it's theirs. That might work, but it sounds complex.

The simpler approach, which is used here, is to place a **TQueue** within each waiter. Since a reference to that waiter is part of the order, all the chef needs to do is fetch the reference and put the filled order on the waiter's **TQueue** – notice how much simpler this is than the other approach.

Finally, to shut everything down, the restaurant keeps track of the number of orders that are filled in the **putOrder()** method, and when it notices that this has exceeded the maximum number of orders (specified on the command line), it calls **shutdown()**, which goes through and calls **interrupt()** for all the threads. The chefs and waiters, if they are waiting on a **get()**, will be interrupted out of that **wait()**.

The only problem with using **interrupt()** to terminate the program is that you'll see that – although the program doesn't hang, and always terminates without exceptions – not all the orders that are placed make it all the way through to completion. One way to solve this problem is to produce a scheme that will send a kind of signal throughout the system, through the **TQueue**s so that anyone waiting on a **get()** will wake up and get the signal. This signal could be a special kind of **Order3** object, or it might be a **null** reference. That exercise will be left to the reader.

Exercise 13

```
//: c13:E13_Restaurant5.java
// {Args: 40}
/****************** Exercise 13 ************************
 * Modify the previous exercise to add Customer objects
 * that are also threads. The Customers will place order
 * requests with WaitPersons, who give the requests to
 * the Chefs, who fulfill the orders and notify the
 * appropriate WaitPerson, who gives it to the
 * appropriate Customer.
 ********************************************************/
class Order4 {
  private static int i = 0;
  private int count = i++;
  private Customer customer;
  private WaitPerson4 waiter;
  public Order4(Customer origin) {
    customer = origin; // Customer that placed the order
  }
  public void setWaiter(WaitPerson4 waiter) {
    this.waiter = waiter;
  }
  // Who should deliver this meal?
  public WaitPerson4 getWaiter() { return waiter; }
  // Who does the meal go to?
  public Customer getCustomer() { return customer; }
  public String toString() {
    return "Order " + count + ", waiter " + waiter
      + ", customer " + customer;
  }
  public int number() { return count; }
}

class Customer extends Thread {
  private static int i = 0;
  private int count = i++;
  private E13_Restaurant5 restaurant;
  private Order4 meal;
  public Customer(E13_Restaurant5 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(!isInterrupted()) {
      // wait()s until a waiter is available:
      WaitPerson4 waiter = restaurant.getWaiter();
      waiter.giveOrder(new Order4(this));
      // Waits until the meal shows up:
      synchronized(this) {
        try {
          wait();
        } catch(InterruptedException e) {
          return; // Quit
        }
      }
      System.out.println(this + " eats " + meal);
      meal = null;
    }
    System.out.println(this + " quitting");
  }
  public synchronized void putMeal(Order4 order) {
    if(meal != null)
      throw new RuntimeException("Already have a meal!");
    meal = order;
    notify();
  }
  public String toString() { return "Customer " + count;}
}

class WaitPerson4 extends Thread {
  private static int i = 0;
  private int count = i++;
  private E13_Restaurant5 restaurant;
  private TQueue meals = new TQueue();
  public WaitPerson4(E13_Restaurant5 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(!isInterrupted()) {
      // Make yourself available to customers:
      restaurant.waiterAvailable(this);
      // Wait for chef to finish a meal:
      Order4 order = (Order4)meals.get();
      if(order == null) // get() returned via interrupt
        break; // Out of while loop
      System.out.println(this + " delivers " + order);
      order.getCustomer().putMeal(order);
    }
    System.out.println(this + " quitting");
  }
  public synchronized void giveOrder(Order4 order) {
    order.setWaiter(this);
    restaurant.putOrder(order);
  }
  public void putMeal(Order4 order) { meals.put(order); }
  public String toString() { return "WaitPerson " + count;}
}

class Chef4 extends Thread {
  private static int i = 0;
  private int count = i++;
  private E13_Restaurant5 restaurant;
  public Chef4(E13_Restaurant5 r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(!isInterrupted()) {
      // TQueue waits until there's an order:
      Order4 order = restaurant.getOrder();
      if(order == null) // get() returned via interrupt
        break; // Out of while loop
      System.out.println(this + " got " + order);
      // Create the meal
      // ...
      // Put the order back on the queue of the waiter
      // that placed it:
      order.getWaiter().putMeal(order);
    }
    System.out.println(this + " quitting");
  }
  public String toString() { return "Chef " + count;}
}

public class E13_Restaurant5 {
  private TQueue orders = new TQueue();
  private TQueue waiterQueue = new TQueue();
  private Chef4[] chefs = new Chef4[3];
  private WaitPerson4[] waiters = new WaitPerson4[6];
  private Customer[] customers = new Customer[12];
  private int maxOrders;
  public E13_Restaurant5(int maxOrders) {
    this.maxOrders = maxOrders;
    for(int i = 0; i < chefs.length; i++)
      chefs[i] = new Chef4(this);
    for(int i = 0; i < waiters.length; i++)
      waiters[i] = new WaitPerson4(this);
    for(int i = 0; i < customers.length; i++)
      customers[i] = new Customer(this);
  }
  public void waiterAvailable(WaitPerson4 waiter) {
    waiterQueue.put(waiter);
  }
  public WaitPerson4 getWaiter() {
    return (WaitPerson4)waiterQueue.get();
  }
  public synchronized void putOrder(Order4 newOrder) {
    if(newOrder.number() > maxOrders)
      shutdown();
    else
      orders.put(newOrder);
  }
  public Order4 getOrder() { return (Order4)orders.get(); }
  public synchronized void shutdown() {
    for(int i = 0; i < chefs.length; i++)
      chefs[i].interrupt();
    for(int i = 0; i < waiters.length; i++)
      waiters[i].interrupt();
    for(int i = 0; i < customers.length; i++)
      customers[i].interrupt();
  }
  public static void main(String[] args) {
    if(args.length < 1) {
      System.err.println("argument: number of meals");
      System.exit(1);
    }
    new E13_Restaurant5(Integer.parseInt(args[0]));
  }
} ///:~
```

Basically, we're just inserting another link in the chain created by the previous solution. Now **Order4** must also contain a reference to the customer who placed the order, as well as the waiter who took the order and who delivers the meal.

Each waiter places tells the restaurant they are available, and a customer requests a waiter from this queue in the restaurant. The customer gives their order to the water, who adds the waiter's reference to the order and hands it to the restaurant. As before, a chef picks up the order and fulfills it, then hands it back to the waiter, who delivers it to the customer. Since the customer will only eat one meal at a time, they do not need to receive the meals through a queue, but instead use a single reference and **wait()**/**notify()**.

As an additional (relatively simple) exercise, change the above code to include a separate **Meal** class, so that the order and the meal aren't just the same object.

Exercise 14

```
//: c13:E14_PipedIO2.java
/******************** Exercise 14 *********************
 * Modify PipedIO.java so that Sender reads and sends
 * lines from a text file.
 *****************************************************/
import java.io.*;
import java.util.*;

class Sender extends Thread {
  private Random rand = new Random();
  private PipedWriter out = new PipedWriter();
  private FileInputStream fis;
  public PipedWriter getPipedWriter() {
    return out;
  }
  public void run() {
    try {
      fis = new FileInputStream("E14_PipedIO2.java");
      int c;
      while(true) {
        while((c = fis.read()) != -1) {
          out.write(c);
          out.flush();
          sleep(rand.nextInt(50));
        }
      }
    } catch (Exception e) {
      throw new RuntimeException(e);
    } finally {
      try {
        fis.close();
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

class Receiver extends Thread {
  private PipedReader in;
  public Receiver(Sender sender) throws IOException {
    in = new PipedReader(sender.getPipedWriter());
  }
  public void run() {
    try {
      System.out.println("Read: ");
      while(true) {
        // Blocks until characters are there:
        System.out.print((char)in.read());
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}

public class E14_PipedIO2 {
  public static void main(String[] args) throws Exception {
    Sender sender = new Sender();
    Receiver receiver = new Receiver(sender);
    sender.start();
    receiver.start();
    new Timeout(10000, "\nTerminated...");
  }
} ///:~
```

Exercise 15

```
//: c13:E15_DiningPhilosophers2.java
// {Args: 5 0 deadlock 5}
/******************** Exercise 15 ***********************
 * Change DiningPilosophers.java so that the philosophers
 * just pick the next available chopstick (when a
 * philosopher is done with their chopsticks, they drop
 * them into a bin. When a philosopher wants to eat, they
 * take the next two available chopsticks from the bin).
 * Does this eliminate the possibility of deadlock? Can
 * you re-introduce deadlock by simply reducing the number
 * of available chopsticks?
 ********************************************************/
import java.util.*;

class Chopstick {
  private static int counter = 0;
  private int number = counter++;
  public String toString() {
    return "Chopstick " + number;
  }
}

class ChopstickBin {
  private TQueue bin = new TQueue();
  public Chopstick get() { return (Chopstick)bin.get(); }
  public void put(Chopstick stick) { bin.put(stick); }
}

class Philosopher extends Thread {
  private static Random rand = new Random();
  private static int counter = 0;
  private int number = counter++;
  private ChopstickBin bin;
  static int ponder = 0; // Package access
  public Philosopher(ChopstickBin bin) {
    this.bin = bin;
    start();
  }
  public void think() {
    System.out.println(this + " thinking");
    if(ponder > 0)
      try {
        sleep(rand.nextInt(ponder));
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
  }
  public void eat() {
    // Get one chopstick from the bin
    Chopstick c1 = bin.get();
    System.out.println(this +
      " has " + c1 + " waiting for another one");
    // Get another chopstick from bin
    Chopstick c2 = bin.get();
    System.out.println(this + " has " + c2);
    System.out.println(this + " eating");
    // Put the chopsticks back in bin.
    bin.put(c1);
    bin.put(c2);
  }
  public String toString() {
    return "Philosopher " + number;
  }
  public void run() {
    while(true) {
      think();
      eat();
    }
  }
}

public class E15_DiningPhilosophers2 {
  public static void main(String[] args) {
    if(args.length < 3) {
      System.err.println("usage:\n" +
        "java E15_DiningPhilosophers2 " +
        "numberOfPhilosophers ponderFactor deadlock " +
        "timeout\n" + "A nonzero ponderFactor will " +
        "generate a random sleep time during think().\n" +
        "If deadlock is not the string " +
        "'deadlock', the program will not deadlock.\n" +
        "A nonzero timeout will stop the program after " +
        "that number of seconds.");
      System.exit(1);
    }
    ChopstickBin bin = new ChopstickBin();
    Philosopher[] philosopher =
      new Philosopher[Integer.parseInt(args[0])];
    Philosopher.ponder = Integer.parseInt(args[1]);
    for(int i = 0; i < philosopher.length; i++)
      bin.put(new Chopstick());
    // One additional chopstick guarantees that at least
    // one philosopher can eat without blocking.
    if(!args[2].equals("deadlock"))
      bin.put(new Chopstick());
    for(int i = 0; i < philosopher.length; i++)
      philosopher[i] = new Philosopher(bin);
    if(args.length == 4)
      new Timeout(Integer.parseInt(args[3]) * 1000,
        "Timed out");
  }
} ///:~
```

The **ChopstickBin** is created using a **TQueue**, so it gains all of **TQueue**'s control logic. Now, instead of picking up only the left and right chopsticks, philosophers grab them out of the common bin and are suspended if there aren't enough. The results are the same, however: if you have as many chopsticks as philosophers, you can deadlock, and it seems to happen fairly quickly (well, at least it does on my machine with my OS and the version of the JDK I have installed. I've learned never to say anything deterministic when it comes to threading, especially Java threading). If you add one extra chopstick to the bin, that seems to prevent deadlock.

Exercise 16

```
//: c13:E16_RequestStop.java
/******************** Exercise 16 **********************
 * Inherit a class from java.util.Timer and implement the
 * requestStop( ) method as in Stopping.java. Modify the
 * CanStop class in java.util.Timer so that it inherits
 * from java.util.Timer and so that the run() method
 * becomes part of the TimerTask. Make the requestStop()
 * method handle both possibilities: that the TimerTask has
 * not begun running, and that it has.
 *******************************************************/
import java.util.*;

class CanStop extends Timer {
  private volatile boolean stop = false;
  private int counter = 0;
  private String id;
  public CanStop(int delay, final String id) {
    // Not a daemon thread
    this.id = id;
    schedule(new TimerTask() {
      public void run() {
        while(!stop && counter < 10000) {
          try {
            Thread.sleep(100);
          } catch(InterruptedException e) {
            throw new RuntimeException(e);
          }
          System.out.println(counter++);
        }
        if(stop) System.out.println(id + " detected flag");
      }
    }, delay);
  }
  public synchronized void requestStop() {
    cancel(); // Cancel pending request
    stop = true; // Flag down currently running thread
  }
}
```

```
class Stopper extends Timer {
  public Stopper(
  int delay, final CanStop stoppable, final String id) {
    super(true); // Daemon thread
    schedule(new TimerTask() {
      public void run() {
        System.out.println("Requesting stop on " + id);
        stoppable.requestStop();
      }
    }, delay);
  }
}

public class E16_RequestStop {
  public static void main(String[] args) {
    System.out.println(
      "Requesting stop before task starts");
    final CanStop stoppable = new CanStop(1000, "before");
    new Stopper(900, stoppable, "before");
    System.out.println(
      "Requesting stop after task starts");
    final CanStop stoppable2 = new CanStop(1000, "after");
    new Stopper(2000, stoppable2, "after");
  }
} ///:~
```

From the **java.util.Timer** Java documentation, you can see that to stop a task before it's run, you call **cancel()**. This has no effect on the task after it has begun running, so for that we still need to set a flag, and the task must check that flag.

Exercise 17

```
//: c13:E17_InterruptSimpleThread.java
/********************** Exercise 17 *********************
 * Modify SimpleThread.java so that all threads receive an
 * interrupt() before they are completed.
 *******************************************************/
public class E17_InterruptSimpleThread extends Thread {
  private int countDown = 5000;
  private static int threadCount = 0;
  public E17_InterruptSimpleThread() {
    super("" + ++threadCount); // Store the thread name
    System.out.println("Creating " + this);
    start();
  }
  public String toString() {
    return "#" + getName() + ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(isInterrupted()) {
        System.out.println(this + " Interrupted");
        return;
      }
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    Thread t[] = new Thread[5];
    for(int i = 0; i < 5; i++)
      t[i] = new E17_InterruptSimpleThread();
    for(int i = 0; i < 5; i++)
      t[i].interrupt();
   }
} ///:~
```

The above solution is the very simple approach. You could also get fancier, and put each thread into a **wait()** and then interrupt out of the **wait()** condition, as shown here:

```
//: c13:E17_InterruptSimpleThread2.java
import java.util.*;

public class E17_InterruptSimpleThread2 extends Thread {
  private static int threadCount = 0;
  public E17_InterruptSimpleThread2() {
    super("" + ++threadCount); // Store the thread name
    System.out.println("Creating " + this);
    start();
  }
  public void run() {
    try {
      synchronized(this) {
        wait();
      }
    } catch(InterruptedException e) {
      System.out.println(this + " interrupted");
    }
  }
  public String toString() { return "#" + getName(); }
  public static void main(String[] args) {
    final Thread t[] = new Thread[5];
    for(int i = 0; i < 5; i++)
      t[i] = new E17_InterruptSimpleThread2();
    new Timer(true).schedule(new TimerTask() {
      public void run() {
        for(int i = 0; i < t.length; i++)
          t[i].interrupt();
      }
    }, 1000);
  }
} ///:~
```

Exercise 18

```
//: c13:E18_ProducerConsumer.java
// {Args: 1 200}
/********************** Exercise 18 **********************
 * Solve a single producer, single consumer problem using
 * wait() and notify(). The producer must not overflow the
 * receiver's buffer, which can happen if the producer is
 * faster than the consumer. If the consumer is faster than
 * the producer, then it must not read the same data more
 * than once. Do not assume anything about the relative
 * speeds of the producer or consumer.
*********************************************************/
class Item {
  private static int counter = 0;
  private int id = counter++;
  public String toString() { return "Item " + id; }
}

class Producer extends Thread {
  private int delay = 1000;
  private FlowQueue output;
  public Producer(FlowQueue output, int sleepTime) {
    this.output = output;
    delay = sleepTime;
    start();
  }
  public void run() {
    while(true) {
      output.put(new Item());
      try {
        sleep(delay);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

class Consumer extends Thread {
  private int delay = 1000;
  private FlowQueue input;
  public Consumer(FlowQueue input, int sleepTime) {
    this.input = input;
    delay = sleepTime;
    start();
  }
  public void run() {
    while(true) {
      System.out.println(input.get());
      try {
        sleep(delay);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

public class E18_ProducerConsumer {
  public static void main(String[] args) throws Exception {
    if(args.length < 2) {
     System.out.println("Usage java E18_ProducerConsumer" +
        " <producer sleep time> <consumer sleep time>");
      System.exit(1);
    }
    int producerSleep = Integer.parseInt(args[0]);
    int consumerSleep = Integer.parseInt(args[0]);
    FlowQueue fq = new FlowQueue(100);
    new Producer(fq, producerSleep);
    new Consumer(fq, consumerSleep);
    new Timeout(2000, "Timed Out");
  }
} ///:~
```

Once again, a queue solves this problem best. The previous **TQueue** can be modified to add flow control:

```
//: c13:FlowQueue.java
// A queue for solving flow-control problems.
import java.util.*;

public class FlowQueue {
  private LinkedList queue = new LinkedList();
  private int maxSize;
  public FlowQueue(int maxSize) {
    this.maxSize = maxSize;
  }
  public synchronized void put(Object v) {
    while(queue.size() >= maxSize)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("Interrupted");
      }
    queue.addFirst(v);
    maxSize++;
    notifyAll();
  }
  public synchronized Object get() {
    while(queue.isEmpty())
      try {
        wait();
      } catch(InterruptedException e) {
        return null; // Says "quit now"
      }
    Object returnVal = queue.removeLast();
    maxSize--;
    notifyAll();
    return returnVal;
  }
} ///:~
```

Ensuring that the consumer cannot read a value more than once is automatically taken care of by using a queue. If the producer tries to overfill the queue, the call to **put()** will suspend the producer's thread until the number of elements drops below **maxSize**. Notice how elegant and reusable this solution is.

Try it with different speeds of production and consumption, for example:

```
java E18_ProducerConsumer 1 200
```

```
java E18_ProducerConsumer 200 1
```

Chapter 14

Exercise 1

```
//: c14:E01_SimpleApplet.java
// {RunByHand}
/****************** Exercise 1 *****************
 * Create an applet/application using the Console
 * class as shown in this chapter. Include a text
 * field and three buttons. When you press each
 * button, make some different text appear in the
 * text field.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Button1 extends JApplet {
  JTextField txt = new JTextField(10);
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2"),
    b3 = new JButton("Button 3");
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String label =
        ((JButton)e.getSource()).getText();
      txt.setText(label + " pressed");
    }
  };
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(txt);
    cp.add(b1);
    cp.add(b2);
    cp.add(b3);
    // Actionlisteners can be added after the
    // button is added to the container:
    b1.addActionListener(al);
    b2.addActionListener(al);
    b3.addActionListener(al);
  }
}

public class E01_SimpleApplet {
  public static void main(String[] args) {
    Console.run(new Button1(), 120, 160);
  }
} ///:~
```

The text that appears in the **JTextField** is created from the label on each button.

Exercise 2

```
//: c14:E02_CheckBoxApplet.java
// {RunByHand}
/****************** Exercise 2 ******************
 * Add a check box to the applet created in
 * Exercise 1, capture the event, and insert
 * different text into the text field.
 ***********************************************/
import com.bruceeckel.swing.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class CheckBoxApplet extends JApplet {
  JTextField txt = new JTextField(12);
  JCheckBox check = new JCheckBox("CheckBox");
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2"),
    b3 = new JButton("Button 3");
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String label =
        ((JButton)e.getSource()).getText();
      txt.setText(label + " pressed");
    }
  };
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(txt);
    cp.add(b1);
    cp.add(b2);
    cp.add(b3);
    cp.add(check);
    b1.addActionListener(al);
    b2.addActionListener(al);
    b3.addActionListener(al);
    check.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JCheckBox jcb = (JCheckBox)e.getSource();
        if(jcb.isSelected())
          txt.setText("Checkbox checked");
        else
          txt.setText("Checkbox unchecked");
      }
    });
  }
}

public class E02_CheckBoxApplet {
  public static void main(String args[]) {
    Console.run(new CheckBoxApplet(), 140, 180);
  }
} ///:~
```

Exercise 3

```
//: c14:E03_Password.java
// {RunByHand}
/****************** Exercise 3 ******************
 * Create an applet/application using Console. In
 * the JDK documentation from java.sun.com, find
 * the JPasswordField and add this to the
 * program. If the user types in the correct
 * password, use Joptionpane to provide a success
 * message to the user.
 **********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Password extends JApplet {
  JPasswordField pwd = new JPasswordField(10);
  public void init() {
    final Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JLabel("Type in your password:"));
    cp.add(pwd);
    pwd.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String message = "Incorrect Password";
        JPasswordField pass =
          (JPasswordField)e.getSource();
        if(new String(pass.getPassword()).equals("Blarth"))
          message = "Correct Password";
        JOptionPane.showMessageDialog(
          null,
          message,
          "information",
          JOptionPane.INFORMATION_MESSAGE);
      }
    });
  }
}

public class E03_Password {
  public static void main(String args[]) {
    Console.run(new Password(), 200, 100);
  }
} ///:~
```

Exercise 4

```
//: c14:E04_AllAction.java
// {RunByHand}
/****************** Exercise 4 ******************
 * Create an applet/application using Console,
 * and add all the Swing components that have an
 * addActionListener() method. (Look these up in
 * the JDK documentation from java.sun.com.
 * Hint: Use the index.) Capture their events and
 * display an appropriate message for each inside
 * a text field.
 ***********************************************/
// I made a change to the exercise and added the
// word "Swing." It turns out there are only a
// few Swing components that have this listener.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class AllAction extends JApplet {
  JTextField txt = new JTextField(30);
  JButton b1 = new JButton("Button 1");
  JComboBox jcb = new JComboBox(new String[]{
    "Elements", "To", "Place", "In", "Combobox"
  });
  JFileChooser jfc = new JFileChooser(".");
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(txt);
    cp.add(b1);
    cp.add(jcb);
    cp.add(jfc);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txt.setText("Button pressed");
      }
    });
    txt.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
              JOptionPane.showMessageDialog(
          null,
          "JTexfField ActionListener fired",
          "information",
          JOptionPane.INFORMATION_MESSAGE);
      }
    });
    jcb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txt.setText("JComboBox selected: " +
          jcb.getSelectedItem());
      }
    });
    jfc.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txt.setText(
          "FileChooser ActionListener fired: " +
          jfc.getSelectedFile());
      }
    });
    new Timer(5000, new ActionListener() {
      int i = 0;
      public void actionPerformed(ActionEvent e) {
        txt.setText("Timer Ticked " + i++);
      }
    }).start();
  }
}

public class E04_AllAction {
  public static void main(String args[]) {
    Console.run(new AllAction(), 550, 400);
  }
} ///:~
```

I had to make a change to the exercise description to avoid confusion – I didn't want you to use AWT components (avoid those at all costs; only use Swing!).

The only time the **ActionListener** for the **JTextField** is fired is when you press the "return" key (although I suspect this could be modified – you can probably find out by digging through the docs).

This is actually a fairly fun way to investigate what these components are capable of, and in the ease of use it shows you (again) what a good design Swing is.

One interesting aspect is the **javax.swing.Timer** (notice there's also a very different **java.util.Timer**). I created the timer without capturing the reference, but the garbage collector never seems to collect it, as it would with any other object whose reference was unused. In cases like this the constructor often registers **this** somewhere so the object won't be garbage collected.

Exercise 5

```
//: c14:E05_TypeableButton.java
// {RunByHand}
/****************** Exercise 5 ******************
 * Create an applet/application using Console,
 * with a JButton and a JTextField. Write and
 * attach the appropriate listener so that if the
 * button has the focus, characters typed into it
 * will appear in the JTextField.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class TypeableButton extends JApplet {
  JTextField txt = new JTextField(10);
  JButton b = new JButton("Button 1");
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(txt);
    cp.add(b);
    b.addKeyListener(new KeyAdapter() {
      public void keyTyped(KeyEvent e) {
        txt.setText(txt.getText() + e.getKeyChar());
      }
    });
  }
}

public class E05_TypeableButton {
  public static void main(String args[]) {
    Console.run(new TypeableButton(), 200, 100);
  }
} ///:~
```

Exercise 6

```
//: c14:E06_Everything.java
// {RunByHand}
/****************** Exercise 6 ******************
 * Create an applet/application using Console.
 * Add to the main frame all the components
 * described in this chapter, including menus and
 * a dialog box.
 ***********************************************/
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class MyDialog extends JDialog {
  public MyDialog(JFrame parent) {
    super(parent, "My dialog", true);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JLabel("Here is my dialog"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dispose(); // Closes the dialog
      }
    });
    cp.add(ok);
    setSize(150,125);
  }
}

class Everything extends JApplet {
  BasicArrowButton
    up = new BasicArrowButton(BasicArrowButton.NORTH),
    down = new BasicArrowButton(BasicArrowButton.SOUTH),
    right = new BasicArrowButton(BasicArrowButton.EAST),
    left = new BasicArrowButton(BasicArrowButton.WEST);
  JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };
  JMenuItem[] items = {
    new JMenuItem("Fee"), new JMenuItem("Fi"),
    new JMenuItem("Fo"),  new JMenuItem("Zip"),
    new JMenuItem("Zap"), new JMenuItem("Zot"),
    new JMenuItem("Olly"), new JMenuItem("Oxen"),
    new JMenuItem("Free") };
  String[] flavors = {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  MyDialog dlg = new MyDialog(null);
  static JPanel showBorder(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JTextField(10));
    cp.add(new JButton("Button"));
    cp.add(new JTextArea(5, 10));
    cp.add(new JToggleButton("JToggleButton"));
    cp.add(new JCheckBox("JCheckBox"));
    cp.add(new JRadioButton("JRadioButton"));
    JPanel jp = new JPanel();
    jp.setBorder(new TitledBorder("Directions"));
    jp.add(up);
    jp.add(down);
    jp.add(left);
    jp.add(right);
    cp.add(jp);
    jp = new JPanel();
    jp.setLayout(new GridLayout(2,4));
    jp.add(showBorder(new TitledBorder("Title")));
    jp.add(showBorder(new EtchedBorder()));
    jp.add(showBorder(new LineBorder(Color.blue)));
    jp.add(showBorder(
      new MatteBorder(5,5,30,30,Color.green)));
    jp.add(showBorder(
      new BevelBorder(BevelBorder.RAISED)));
    jp.add(showBorder(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    jp.add(showBorder(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.red))));
    cp.add(jp);
    cp.add(new JScrollPane(
      new JTextArea("", 5, 10),
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS));
    JTextPane jtp =
      new JTextPane(new DefaultStyledDocument());
    jtp.setText("This is a JTextPane");
    cp.add(jtp);
    cp.add(new JComboBox(flavors));
    cp.add(new JScrollPane(new JList(flavors)));
    JTabbedPane tabs = new JTabbedPane();
    for(int i = 0; i < flavors.length; i++)
      tabs.addTab(flavors[i],
        new JButton("Tabbed pane " + i));
    cp.add(tabs);
    JOptionPane.showMessageDialog(null,
      "There's a bug on you!", "Hey!",
      JOptionPane.ERROR_MESSAGE);
    for(int i = 0; i < items.length; i++)
      menus[i%3].add(items[i]);
    JMenuBar mb = new JMenuBar();
    for(int i = 0; i < menus.length; i++)
      mb.add(menus[i]);
    setJMenuBar(mb);
    dlg.show();
    new JFileChooser().showOpenDialog(null);
    JProgressBar pb = new JProgressBar();
    JSlider sb =
      new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    pb.setModel(sb.getModel()); // Share model
    cp.add(pb);
    cp.add(sb);
    cp.add(new JTree(flavors));
    cp.add(new JTable(new String[][] {
      {"one", "two", "three", "four"},
      {"five", "six", "seven", "eight"},
      {"nine", "ten", "eleven", "twelve"},
    }, new String[] { "first", "second",
       "third", "fourth"}));
  }
}

public class E06_Everything {
  public static void main(String args[]) {
    Console.run(new Everything(), 900, 550);
  }
} ///:~
```

This was basically an exercise in moving through the chapter and grabbing code out of most of the examples and pasting it in here. Sometimes you had to go to the JavaDocs and look a few things up.

Exercise 7

```
//: c14:E07_OriginalCase.java
// {RunByHand}
/****************** Exercise 7 ******************
 * Modify TextFields.java so that the characters
 * in t2 retain the original case that they were
 * typed in, instead of automatically being
 * forced to upper case.
 ***********************************************/
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class TextFields extends JApplet {
  JButton
    b1 = new JButton("Get Text"),
    b2 = new JButton("Set Text");
  JTextField
    t1 = new JTextField(30),
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  String s = new String();
  UpperCaseDocument
    ucd = new UpperCaseDocument();
  public void init() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    DocumentListener dl = new T1();
    t1.addActionListener(new T1A());
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t1);
    cp.add(t2);
    cp.add(t3);
  }
  class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
      t2.setText(t1.getText());
      t3.setText("Text: "+ t1.getText());
    }
    public void removeUpdate(DocumentEvent e) {
      t2.setText(t1.getText());
    }
  }
  class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
      t3.setText("t1 Action Event " + count++);
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(t1.getSelectedText() == null)
        s = t1.getText();
      else
        s = t1.getSelectedText();
      t1.setEditable(true);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ucd.setUpperCase(false);
      t1.setText("Inserted by Button 2: " + s);
      ucd.setUpperCase(true);
      t1.setEditable(false);
    }
  }
}

class UpperCaseDocument extends PlainDocument {
  boolean upperCase = false; // true;
  public void setUpperCase(boolean flag) {
    // upperCase = flag;
  }
  public void insertString(int offset,
    String string, AttributeSet attributeSet)
    throws BadLocationException {
      if(upperCase)
        string = string.toUpperCase();
      super.insertString(offset,
        string, attributeSet);
  }
}

public class E07_OriginalCase {
  public static void main(String[] args) {
    Console.run(new TextFields(), 375, 150);
  }
} ///:~
```

This exercise just forces you to study the example thoroughly enough to know where to make the changes, which are the second and fourth lines of **UpperCaseDocument**.

Exercise 8

```
//: c14:E08_LeftToReader.java
/****************** Exercise 8 ******************
 * Locate and download one or more of the free
 * GUI builder development environments available
 * on the Internet, or buy a commercial product.
 * Discover what is necessary to add BangBean to
 * this environment and to use it.
 ***********************************************/
public class E08_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Left to the reader");
  }
} ///:~
```

Exercise 9

```
//: c14:E09_LeftToReader.java
/****************** Exercise 9 ******************
 * Add Frog.class to the manifest file as shown
 * in this chapter and run jar to create a JAR
 * file containing both Frog and BangBean. Now
 * either download and install the Bean Builder from Sun
 * or use your own Beans-enabled program builder
 * tool and add the JAR file to your environment
 * so you can test the two Beans.
 ***********************************************/
public class E09_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Left to the reader");
  }
} ///:~
```

Exercise 10

```
//: c14:valve:E10_Valve.java
// {RunByHand}
/****************** Exercise 10 *****************
 * Create your own JavaBean called Valve that
 * contains two properties: a boolean called "on"
 * and an int called "level." Create a manifest
 * file, use jar to package your Bean, then load
 * it into the Bean Builder or into a Beans-enabled
 * program builder tool so that you can test it.
***********************************************/
// To test the program, run the following command
// in directory c14:
// java c14.valve.E10_Valve
// To make the jar file, run the following command
// in the directory above c14:
// jar cfm E10_Valve.jar E10_Valve.mf c14/valve
package c14.valve;

public class E10_Valve {
  private boolean on = false;
  private int level = 0;
  public E10_Valve() {}
  public E10_Valve(boolean on, int level) {
    this.on = on;
    this.level = level;
  }
  public boolean isOn() { return on; }
  public void setOn(boolean on) { this.on = on; }
  public int getLevel() { return level; }
  public void setLevel(int level) { this.level = level; }
  public static void main(String args[]) {
    E10_Valve v = new E10_Valve(true, 100);
    System.out.println("v.isOn() = " + v.isOn());
    System.out.println("v.getLevel() = " + v.getLevel());
  }
} ///:~
```

Note the organization of the file inside its own package. Here is the manifest file:

```
//:! :E10_Valve.mf
Manifest-Version: 1.0

Name: valve/E10_Valve.class
Java-Bean: True
///:~
```

You must run the **jar** command from the root of the code tree using the commands shown as comments in **E10_Valve.java**.

To validate the resulting **jar** file, load it into into your Beans-enabled editor (I used the freely-downloadable version of Borland's JBuilder).

Exercise 11

```
//: c14:E11_MessageAction.java
// {RunByHand}
/****************** Exercise 11 *****************
 * Modify MessageBoxes.java so that it has an
 * individual ActionListener for each button
 * (instead of matching the button text).
 ***********************************************/
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class MessageBoxes extends JApplet {
  JButton
    b1 = new JButton("Alert"),
    b2 = new JButton("Yes/No"),
    b3 = new JButton("Color"),
    b4 = new JButton("Input"),
    b5 = new JButton("3 Vals");
  JTextField txt = new JTextField(15);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(b3);
    cp.add(b4);
    cp.add(b5);
    cp.add(txt);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
          "There's a bug on you!", "Hey!",
          JOptionPane.ERROR_MESSAGE);
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showConfirmDialog(null,
          "or no", "choose yes",
          JOptionPane.YES_NO_OPTION);
      }
    });
    b3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Object[] options = { "Red", "Green" };
        int sel = JOptionPane.showOptionDialog(
          null, "Choose a Color!", "Warning",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          options, options[0]);
          if(sel != JOptionPane.CLOSED_OPTION)
            txt.setText(
              "Color Selected: " + options[sel]);
      }
    });
    b4.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String val = JOptionPane.showInputDialog(
            "How many fingers do you see?");
        txt.setText(val);
      }
    });
    b5.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Object[] selections = {
          "First", "Second", "Third" };
        Object val = JOptionPane.showInputDialog(
          null, "Choose one", "Input",
          JOptionPane.INFORMATION_MESSAGE,
          null, selections, selections[0]);
        if(val != null)
          txt.setText(
            val.toString());
      }
    });
  }
}

public class E11_MessageAction {
  public static void main(String args[]) {
    Console.run(new MessageBoxes(), 200, 200);
  }
} ///:~
```

I used anonymous inner classes for the **ActionListener**s here, but you could also have used the other approaches.

Exercise 12

```
//: c14:E12_NewEvent.java
// {RunByHand}
/****************** Exercise 12 *****************
 * Monitor a new type of event in TrackEvent.java
 * by adding the new event handling code. You'll
 * need to discover on your own the type of event
 * that you want to monitor.
 ***********************************************/
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class TrackEvent extends JApplet {
  HashMap h = new HashMap();
  String[] event = {
    "actionPerformed", "stateChanged",
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  MyButton
    b1 = new MyButton(Color.blue, "test1"),
    b2 = new MyButton(Color.red, "test2");
  class MyButton extends JButton {
    void report(String field, String msg) {
      ((JTextField)h.get(field)).setText(msg);
    }
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        report("actionPerformed", e.paramString());
      }
    };
    ChangeListener cl = new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        report("stateChanged", e.toString());
      }
    };
    FocusListener fl = new FocusListener() {
      public void focusGained(FocusEvent e) {
        report("focusGained", e.paramString());
      }
      public void focusLost(FocusEvent e) {
        report("focusLost", e.paramString());
      }
    };
    KeyListener kl = new KeyListener() {
      public void keyPressed(KeyEvent e) {
        report("keyPressed", e.paramString());
      }
      public void keyReleased(KeyEvent e) {
        report("keyReleased", e.paramString());
      }
      public void keyTyped(KeyEvent e) {
        report("keyTyped", e.paramString());
      }
    };
    MouseListener ml = new MouseListener() {
      public void mouseClicked(MouseEvent e) {
        report("mouseClicked", e.paramString());
      }
      public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e.paramString());
      }
      public void mouseExited(MouseEvent e) {
        report("mouseExited", e.paramString());
      }
      public void mousePressed(MouseEvent e) {
        report("mousePressed", e.paramString());
      }
      public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e.paramString());
      }
    };
    MouseMotionListener mml =
      new MouseMotionListener() {
```

```
      public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e.paramString());
      }
      public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e.paramString());
      }
    };
    public MyButton(Color color, String label) {
      super(label);
      setBackground(color);
      addFocusListener(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
      addActionListener(al);
      addChangeListener(cl);
    }
  }
  public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i < event.length; i++) {
      JTextField t = new JTextField();
      t.setEditable(false);
      c.add(new JLabel(event[i], JLabel.RIGHT));
      c.add(t);
      h.put(event[i], t);
    }
    c.add(b1);
    c.add(b2);
  }
}

public class E12_NewEvent {
  public static void main(String[] args) {
    Console.run(new TrackEvent(), 700, 500);
  }
} ///:~
```

You can just go to **JButton** and see what kind of “add listener” methods it has. An obvious one is **addActionListener( )**, but I also added one for **addChangeListener( )**.

Exercise 13

```
//: c14:E13_RandomColorButton.java
// {RunByHand}
/****************** Exercise 13 *****************
 * Inherit a new type of button from JButton.
 * Each time you press this button, it should
 * change its color to a randomly-selected value.
 * See ColorBoxes.java in Chapter 14 for an
 * example of how to generate a random color
 * value.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class RandomColorButton extends JButton {
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  public RandomColorButton(String text) {
    super(text);
    setBackground(newColor());
    addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        setBackground(newColor());
      }
    });
  }
}

public class
E13_RandomColorButton extends JApplet {
  public void init() {
    Container c = getContentPane();
    c.setLayout(new BorderLayout());
    c.add(new RandomColorButton("Random Colors"));
  }
  public static void main(String args[]) {
    Console.run(
      new E13_RandomColorButton(), 150, 75);
  }
} ///:~
```

Exercise 14

```
//: c14:E14_UseTextArea.java
// {RunByHand}
/****************** Exercise 14 *****************
 * Modify TextPane.java to use a JTextArea
 * instead of a JTextPane.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

class TextPane extends JApplet {
  JButton b = new JButton("Add Text");
  // Only needed to change this one line:
  JTextArea tp = new JTextArea();
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() +
            sg.next() + "\n");
      }
    });
    Container cp = getContentPane();
    cp.add(new JScrollPane(tp));
    cp.add(BorderLayout.SOUTH, b);
  }
}

public class E14_UseTextArea {
  public static void main(String[] args) {
    Console.run(new TextPane(), 475, 425);
  }
} ///:~
```

From the example copied in from the book, I only had to change the line that defined the **JTextPane**. This is one of the strengths of object-oriented programming, when you're using classes from the same hierarchy – they have the same interface.

Exercise 15

```
//: c14:E15_RadioMenus.java
// {RunByHand}
/****************** Exercise 15 *****************
 * Modify Menus.java to use radio buttons instead
 * of check boxes on the menus.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class RadioMenus extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  JTextField t = new JTextField("No flavor", 30);
  JMenuBar mb1 = new JMenuBar();
  JMenu
    f = new JMenu("File"),
    m = new JMenu("Flavors"),
    s = new JMenu("Safety");
  // Alternative approach:
  JRadioButtonMenuItem[] safety = {
    new JRadioButtonMenuItem("Guard"),
    new JRadioButtonMenuItem("Hide")
  };
  JMenuItem[] file = {
    new JMenuItem("Open"),
  };
  // A second menu bar to swap to:
  JMenuBar mb2 = new JMenuBar();
  JMenu fooBar = new JMenu("fooBar");
  JMenuItem[] other = {
    // Adding a menu shortcut (mnemonic) is very
    // simple, but only JMenuItems can have them
    // in their constructors:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // No shortcut:
    new JMenuItem("Baz"),
  };
  JButton b = new JButton("Swap Menus");
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuBar m = getJMenuBar();
      setJMenuBar(m == mb1 ? mb2 : mb1);
      validate(); // Refresh the frame
    }
  }
  class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      String actionCommand =
        target.getActionCommand();
      if(actionCommand.equals("Open")) {
        String s = t.getText();
        boolean chosen = false;
        for(int i = 0; i < flavors.length; i++)
          if(s.equals(flavors[i])) chosen = true;
        if(!chosen)
          t.setText("Choose a flavor first!");
        else
          t.setText("Opening "+ s +". Mmm, mm!");
      }
    }
  }
  class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      t.setText(target.getText());
    }
  }
  // Alternatively, you can create a different
  // class for each different MenuItem. Then you
  // Don't have to figure out which one it is:
  class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Foo selected");
    }
  }
  class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Bar selected");
    }
  }
  class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Baz selected");
    }
  }
  class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      JRadioButtonMenuItem target =
        (JRadioButtonMenuItem)e.getSource();
      String actionCommand =
        target.getActionCommand();
      if(actionCommand.equals("Guard"))
        t.setText("Guard the Ice Cream! " +
          "Guarding is " + target.isSelected());
      else if(actionCommand.equals("Hide"))
        t.setText("Hide the Ice Cream! " +
          "Is it cold? " + target.isSelected());
    }
  }
  public void init() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[0].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < flavors.length; i++) {
      JMenuItem mi = new JMenuItem(flavors[i]);
      mi.addActionListener(fl);
      m.add(mi);
      // Add separators at intervals:
      if((i+1) % 3 == 0)
        m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++)
      s.add(safety[i]);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < file.length; i++) {
      file[i].addActionListener(fl);
      f.add(file[i]);
    }
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.add(t, BorderLayout.CENTER);
    // Set up the system for swapping menus:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    cp.add(b, BorderLayout.NORTH);
    for(int i = 0; i < other.length; i++)
      fooBar.add(other[i]);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
  }
}

public class E15_RadioMenus {
  public static void main(String args[]) {
    Console.run(new RadioMenus(), 300, 100);
  }
} ///:~
```

Once the example was copied in, all **JCheckBoxMenuItem**s were replaced with **JRadioButtonMenuItem**s. In addition, calls to **getState( )** were replaced with calls to **isSelected( )**.

Exercise 16

```
//: c14:E16_SimplifyList.java
// {RunByHand}
/****************** Exercise 16 *****************
 * Simplify List.java by passing the array to the
 * constructor and eliminating the dynamic
 * addition of elements to the list.
 ***********************************************/
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;
```

```
class SimplifyList extends JApplet {
  DefaultListModel lItems=new DefaultListModel();
  JList lst = new JList(new String[] {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  });
  JTextArea t = new JTextArea(
    lst.getModel().getSize(), 20);
  ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(
        ListSelectionEvent e) {
          t.setText("");
          Object[] items=lst.getSelectedValues();
          for(int i = 0; i < items.length; i++)
            t.append(items[i] + "\n");
        }
    };
  int count = 0;
  public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());
    // Create Borders for components:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.black);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Add items to the Content Pane for Display
    cp.add(t);
    cp.add(lst);
    // Register event listener
    lst.addListSelectionListener(ll);
  }
}

public class E16_SimplifyList {
  public static void main(String args[]) {
    Console.run(new SimplifyList(), 250, 375);
  }
} ///:~
```

Most of this example involved chopping out the extra code. However, you also had to understand that to access information about the elements in the list you must use the list model, accessed by calling **getModel( )**.

Exercise 17

```
//: c14:E17_SineDrawBean.java
// {RunByHand}
/****************** Exercise 17 *****************
 * Modify SineWave.java to turn SineDraw into a
 * JavaBean by adding "getter" and "setter"
 * methods.
 ***********************************************/
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class SineDrawBean extends JPanel {
  static final int SCALEFACTOR = 200;
  int cycles;
  int points;
  double[] sines;
  int[] pts;
  SineDrawBean() { setCycles(5); }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI/SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    repaint();
  }
  public int getCycles() {
    return cycles;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int maxWidth = getWidth();
    double hstep = (double)maxWidth/(double)points;
    int maxHeight = getHeight();
    pts = new int[points];
    for(int i = 0; i < points; i++)
      pts[i] = (int)(sines[i] * maxHeight/2 * .95
                     + maxHeight/2);
    g.setColor(Color.red);
    for(int i = 1; i < points; i++) {
      int x1 = (int)((i - 1) * hstep);
      int x2 = (int)(i * hstep);
      int y1 = pts[i-1];
      int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
}

class SineDrawApplet extends JApplet {
  SineDrawBean sines = new SineDrawBean();
  JSlider adjustCycles = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(sines);
    adjustCycles.addChangeListener(
      new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
          sines.setCycles(
            ((JSlider)e.getSource()).getValue());
        }
      }
    );
    cp.add(BorderLayout.SOUTH, adjustCycles);
  }
}

public class E17_SineDrawBean {
  public static void main(String args[]) {
    Console.run(new SineDrawApplet(), 700, 400);
  }
} ///:~
```

This is a bit of a trick question. Notice that the only method I added was **getCycles( )**, and I did not add getters and setters for **SCALEFACTOR**, **points**, **sines** or **pts**. That is because those are *dependendent variables* – they are calculated within the object based on other values. Thus, you wouldn't want to allow those values to be set from outside of the bean, and they aren't of much interest to client programmers of the bean.

You have to be careful this way when designing beans. Some people think that getters and setters are trivial access methods to a variable inside an object, but for example **setCycles( )** does much more than modify **cycles**.

Exercise 18

```
//: c14:E18_SketchBox.java
// {RunByHand}
/****************** Exercise 18 *****************
 * Remember the "sketching box" toy with two
 * knobs, one that controls the vertical movement
 * of the drawing point, and one that controls
 * the horizontal movement? Create one of those,
 * using SineWave.java to get you started.
 * Instead of knobs, use sliders. Add a button
 * that will erase the entire sketch.
 ***********************************************/
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class SketchArea extends JPanel {
  java.util.List points = new ArrayList();
  class Point {
    int x, y;
    public Point(int x, int y) {
      this.x = x;
      this.y = y;
    }
  }
  public void addPoint(int x, int y) {
    points.add(new Point(x,y));
    repaint();
  }
  public void clear() {
    points.clear();
    repaint();
  }
  Point previousPoint;
  void drawPoint(Graphics g, Point p) {
    // So that it starts from anywhere, rather
    // than drawing from 0, 0:
    if(previousPoint == null) {
      previousPoint = p;
      return;
    }
    g.drawLine(previousPoint.x, previousPoint.y,
      p.x, p.y);
    previousPoint = p;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.red);
    previousPoint = null;
    Iterator it = points.iterator();
    while(it.hasNext())
      drawPoint(g, (Point)it.next());
  }
}

class SketchBox extends JApplet {
  SketchArea sketch = new SketchArea();
  JSlider
    hAxis = new JSlider(),
    vAxis = new JSlider(JSlider.VERTICAL);
  JButton erase = new JButton("Erase");
  ChangeListener cl = new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        sketch.addPoint(
          hAxis.getValue(), vAxis.getValue());
        erase.setText(
          "[Erase]   points.size() = " +
          sketch.points.size());
      }
    };
  public void init() {
    Container cp = getContentPane();
    cp.add(sketch);
    hAxis.setValue(0);
    vAxis.setValue(0);
    // So vertical axis synchronizes with line:
    vAxis.setInverted(true);
    hAxis.addChangeListener(cl);
    vAxis.addChangeListener(cl);
    cp.add(BorderLayout.SOUTH, hAxis);
    cp.add(BorderLayout.WEST, vAxis);
    cp.add(BorderLayout.NORTH, erase);
    erase.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        sketch.clear();
        erase.setText(
          "[Erase]   points.size() = " +
          sketch.points.size());
      }
    });
    // The width and height values are zero until
    // initial resizing (when the component is
    // first drawn). Also takes care of things
    // if it's resized:
    addComponentListener(new ComponentAdapter() {
      public void componentResized(ComponentEvent e) {
        super.componentResized(e);
        hAxis.setMaximum(sketch.getWidth());
        vAxis.setMaximum(sketch.getHeight());
      }
    });
  }
}
```

```
public class E18_SketchBox {
  public static void main(String args[]) {
    Console.run(new SketchBox(), 700, 400);
  }
} ///:~
```

SketchArea contains a **List** of **Point** objects, each of which has an **x** and **y** coordinate. When **paintComponent( )** is called, it moves through the list and draws (using **drawPoint( )**) a line from each previous point to the current point.

The **SketchBox** contains a horizontal **JSlider** (the default orientation) and a vertical one. The **ChangeListener** for both **JSlider**s just adds a new **Point** at the current location (it also changes the text on the **JButton** so we can see how many points are used in a particular drawing).

In **init( )**, note the use of **setInverted( )** on **vAxis** – without this the line is drawn in the opposite direction to the motion of the slider.

The other trick is the use of **addComponentListener( )**, passing it a **ComponentAdapter** with an overriden **componentResized( )**. This isn't just an enhancement so that the program responds properly to being resized (which it does). You can't just call **setMaximum( )** in **init( )**, because **getWidth( )** and **getHeight( )** will always return zero in **init( )** – the applet hasn't been sized yet during **init( )**. When the applet is given its initial size, *after* **init( )** is finished, then **componentResized( )** is called and it can set the proper maximums for the sliders.

Exercise 19

```
//: c14:E19_AnimatedSine.java
// {RunByHand}
/****************** Exercise 19 ******************
 * Starting with SineWave.java, create a program
 * (an applet/application using the Console class)
 * that draws an animated sine wave that appears
 * to scroll past the viewing window like an
 * oscilloscope, driving the animation with a
 * Thread. The speed of the animation should be
 * controlled with a java.swing.JSlider control.
 ********************************************/
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class SineDraw extends JPanel implements Runnable {
  static final int SCALEFACTOR = 200;
  int cycles;
  int points;
  double[] sines;
  int[] pts;
  Thread self;
  int delay = 50;
  JSlider speed = new JSlider(0, 100, 50);
  double offset = 0;
  SineDraw() {
    super(new BorderLayout());
    setCycles(5);
    self = new Thread(this);
    self.start();
    speed.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        delay = speed.getValue();
      }
    });
    add(BorderLayout.SOUTH, speed);
    // So that left to right is slow to fast:
    speed.setInverted(true);
  }
  public synchronized
  void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    pts = new int[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI/SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    repaint();
  }
  public void run() {
    while(true) {
      offset += 0.25;
      synchronized(this) {
        for(int i = 0; i < points; i++) {
          double radians =
            (Math.PI/SCALEFACTOR) * i + offset;
          sines[i] = Math.sin(radians);
        }
      }
      repaint();
      try {
        Thread.sleep(delay);
      } catch(InterruptedException e) {
        System.out.println(
          "InterruptedException = " + e);
      }
    }
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.red);
    int maxWidth = getWidth();
    double hstep =(double)maxWidth/(double)points;
    int maxHeight = getHeight();
    for(int i = 0; i < points; i++)
      // Some adjustments here to compensate for
      // the added JSlider in the panel:
      pts[i] = (int)(sines[i] * maxHeight/2 * .89
                      + maxHeight/2 * .91);
    for(int i = 1; i < points; i++) {
      int x1 = (int)((i - 1) * hstep);
      int x2 = (int)(i * hstep);
      int y1 = pts[i-1];
      int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
```

```
}

class SineWave extends JApplet {
  SineDraw sines = new SineDraw();
  JSlider cycles = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(sines);
    cycles.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        sines.setCycles(
          ((JSlider)e.getSource()).getValue());
      }
    });
    cp.add(BorderLayout.SOUTH, cycles);
  }
}

public class E19_AnimatedSine {
  public static void main(String[] args) {
    Console.run(new SineWave(), 700, 400);
  }
} ///:~
```

The primary changes were made to **SineDraw**, which is now **Runnable**, with a **run( )** method that drives the animation. In the constructor, a **BorderLayout** is used instead of the default **FlowLayout** and the **JSlider speed** is added. When **speed** is changed, the **delay** is changed; this is used in **run( )** as the **sleep( )** delay.

The **run( )** method calculates the points as before, except that a value **offset** is incremented and added each time – this is the value that moves the sinewave forward.

Note the use of the **synchronized** clause in **run( )** and that **setCycles( )** is synchronized – this prevents threading collision problems when the array sizes are being changed.

Exercise 20

```
//: c14:E20_MultipleSine.java
```

```
// {RunByHand}
/****************** Exercise 20 *****************
 * Modify Exercise 19 so that multiple sine wave
 * panels are created within the application.
 * The number of sine wave panels should be
 * controlled by HTML tags or command-line
 * parameters.
 ***********************************************/
import com.bruceeckel.swing.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

class MultiSineWave extends JApplet {
  boolean isApplet = true;
  SineDraw[] sines;
  int panels = 1;
  JSlider cycles = new JSlider(1, 30, 5);
  public void init() {
    if(isApplet) {
      String sz = getParameter("panels");
      if(sz != null)
        panels = Integer.parseInt(sz);
    }
    Container cp = getContentPane();
    int side = Math.round(
      (float)Math.sqrt((double)panels));
    JPanel jp =
      new JPanel(new GridLayout(side, side));
    sines = new SineDraw[panels];
    for(int i = 0; i < sines.length; i++) {
      sines[i] = new SineDraw();
      jp.add(sines[i]);
    }
    cp.add(jp);
    cp.add(BorderLayout.SOUTH, cycles);
    cycles.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        for(int i = 0; i < sines.length; i++)
          sines[i].setCycles(
            ((JSlider)e.getSource()).getValue());
      }});
  }
}

public class E20_MultipleSine {
  public static void main(String[] args) {
    MultiSineWave applet = new MultiSineWave();
    applet.isApplet = false;
    if(args.length != 0)
      applet.panels = Integer.parseInt(args[0]);
    else
      applet.panels = 4;
    Console.run(applet, 700, 400);
  }
} ///:~
```

The **SineDraw** class from the previous exercise is used here. Much of this solution is just the usual bookkeeping of getting the argument from the command line or applet (or just using a default value of 4), and laying out the panels. Here, an array of **SineDraw** objects is used, and when you change the number of cycles the **ChangeListener** moves through the array and calls **setCycles( )** for each **SineDraw** object.

To create a reasonably square array, I take the square root of the desired number of **SineDraw** objects, round it up, and use it as both dimensions of a **GridLayout**.

Exercise 21

```
//: c14:E21_TimerAnimation.java
// {RunByHand}
/********************** Exercise 21 ********************
 * Modify Exercise 19 so that the java.swing.Timer class
 * is used to drive the animation. Note the difference
 * between this and java.util.Timer.
 ******************************************************/
import com.bruceeckel.swing.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

class SineDraw_T extends JPanel {
  static final int SCALEFACTOR = 200;
  int cycles;
  int points;
  double[] sines;
  int[] pts;
  JSlider speed = new JSlider(0, 100, 50);
  double offset = 0;
  int delay = 50;
  Timer timer = new Timer(delay,
    new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        offset += 0.25;
        for(int i = 0; i < points; i++) {
          double radians = (Math.PI/SCALEFACTOR)
            * i + offset;
          sines[i] = Math.sin(radians);
          repaint();
        }
      }
    });
  SineDraw_T() {
    super(new BorderLayout());
    setCycles(5);
    speed.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        timer.setDelay(speed.getValue());
      }
    });
    add(BorderLayout.SOUTH, speed);
    speed.setInverted(true);
    timer.start();
  }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    pts = new int[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI/SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.red);
    int maxWidth = getWidth();
    double hstep =(double)maxWidth/(double)points;
    int maxHeight = getHeight();
    for(int i = 0; i < points; i++)
      // Some adjustments here to compensate for
      // the added JSlider in the panel:
      pts[i] = (int)(sines[i] * maxHeight/2 * .94
                     + maxHeight/2 * .96);
    for(int i = 1; i < points; i++) {
      int x1 = (int)((i - 1) * hstep);
      int x2 = (int)(i * hstep);
      int y1 = pts[i-1];
      int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
}

class SineWave_T extends JApplet {
  SineDraw_T sines = new SineDraw_T();
  JSlider cycles = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(sines);
    cycles.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        sines.setCycles(
          ((JSlider)e.getSource()).getValue());
      }
    });
    cp.add(BorderLayout.SOUTH, cycles);
  }
}
```

```
public class E21_TimerAnimation {
  public static void main(String args[]) {
    Console.run(new SineWave_T(), 700, 400);

  }
} ///:~
```

javax.swing.Timer repeatedly calls an **ActionListener**. The code here is simpler than in Exercise 20.

Exercise 22

```
//: c14:E22_WindozeProgress.java
// {RunByHand}
/****************** Exercise 22 *****************
 * Create an "asymptotic progress indicator" that
 * gets slower and slower as it approaches the
 * finish point. Add random erratic behavior so
 * it will periodically look like it's starting
 * to speed up.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class WindozeProgress extends JApplet {
  JProgressBar pb = new JProgressBar(0, 100);
  JLabel jl =
    new JLabel("Makin' some progress now!");
  Timer tm = new Timer(50, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(pb.getValue() == pb.getMaximum()) {
        tm.stop();
        jl.setText("That wasn't so bad, was it?");
      }
      double percent =
        (double)pb.getValue() /
        (double)pb.getMaximum();
      tm.setDelay((int)(tm.getDelay() *
        (1.0 + percent * 0.07)));
      pb.setValue(pb.getValue() + 1);
      if(percent > 0.90)
        if(Math.random() < 0.25)
          pb.setValue(pb.getValue() - 10);
    }
  });
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(pb);
    cp.add(jl);
    pb.setValue((int)(pb.getMaximum() * 0.25));
    tm.start();
  }
}

public class E22_WindozeProgress {
  public static void main(String args[]) {
    Console.run(new WindozeProgress(), 300, 100);
  }
} ///:~
```

A **Timer** is used to control the progress bar, and the delay on the timer is adjusted according to how close you're getting to being finished. If you're higher than 90%, a random number is chosen and if it's less than 0.25 (in a range of 0-1) then the progress bar is decremented by 10 units.

The trick is to adjust these values so that they mess with the user's head. I'm convinced they spend a lot of time at MicroSquish doing this.

Exercise 23

```
//: c14:E23_SharedListener.java
// {RunByHand}
/****************** Exercise 23 *****************
 * Modify Progress.java so that it does not share
 * models, but instead uses a listener to connect
 * the slider and progress bar.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

class Progress extends JApplet {
  JProgressBar pb = new JProgressBar();
  JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(2,1));
    cp.add(pb);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Slide Me"));
//    pb.setModel(sb.getModel()); // Share model
    cp.add(sb);
    sb.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        pb.setValue(sb.getValue());
      }
    });
  }
}

public class E23_SharedListener {
  public static void main(String args[]) {
    Console.run(new Progress(), 300, 200);
  }
} ///:~
```

I only commented out the **setModel( )** call and added the **ChangeListener**.

Exercise 24

```
//: c14:E24_LeftToReader.java
/****************** Exercise 24 *****************
```

```
 * Follow the instructions in the section titled “Packaging
 * an applet into a JAR file” to place TicTacToe.java into
 * a JAR file. Create an HTML page with the simple version
 * of the applet tag along with the archive specification
 * to use the JAR file. Run HTMLconverter on file to
 * produce a working HTML file.
 ***********************************************/
public class E24_LeftToReader {
  public static void main(String args[]) {
    System.out.println("Left to Reader");
  }
} ///:~
```

Exercise 25

```
//: c14:E25_ColorMixer.java
// {RunByHand}
/****************** Exercise 25 *****************
 * Create an applet/application using Console.
 * This should have three sliders, one each for
 * the red, green, and blue values in
 * java.awt.Color. The rest of the form should be
 * a JPanel that displays the color determined by
 * the three sliders. Also include non-editable
 * text fields that show the current RGB values.
 ***********************************************/
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class ColorMixer extends JApplet {
  class ColorSlider extends JPanel {
    JSlider slider;
    JTextField value;
    public ColorSlider(String title) {
      super(new FlowLayout());
      slider = new JSlider(0, 255, 0);
      value = new JTextField(3);
      add(new JLabel(title));
      add(slider);
      add(value);
      slider.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
          value.setText("" + slider.getValue());
          reColor();
        }
      });
    }
  }
  ColorSlider
    red = new ColorSlider("red"),
    green = new ColorSlider("green"),
    blue = new ColorSlider("blue");
  JPanel color = new JPanel();
  void reColor() {
    color.setBackground(new Color(
      red.slider.getValue(),
      green.slider.getValue(),
      blue.slider.getValue()));
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(2, 1));
    JPanel jp = new JPanel(new FlowLayout());
    jp.add(red);
    jp.add(green);
    jp.add(blue);
    cp.add(jp);
    cp.add(color);
    reColor();
  }
}

public class E25_ColorMixer {
  public static void main(String args[]) {
    Console.run(new ColorMixer(), 350, 300);
  }
} ///:~
```

The **ColorSlider** inner class really simplifies and organizes the code well.

Additional exercise: add a check box which, when checked, causes the sliders to jump to the next web-safe color when you release them.

Exercise 26

```
//: c14:E26_ColorChooser.java
package c14; /* Added by Eclipsify */
// {RunByHand}
/****************** Exercise 26 *****************
 * In the JDK documentation for javax.swing,
 * look up the JColorChooser. Write a program
 * with a button that brings up the color
 * chooser as a dialog.
 ***********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class ColorChooser extends JApplet {
  JButton b1 = new JButton("Color Chooser");
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JColorChooser.showDialog(
          null, "E26_ColorChooser", Color.cyan);
      }
    });
  }
}

public class E26_ColorChooser {
  public static void main(String args[]) {
    Console.run(new ColorChooser(), 150, 75);
  }
} ///:~
```

Additional exercise: use the color display panel from Exercise 22 to show the resulting color after the dialog is closed.

Exercise 27

```
//: c14:E27_Cursors.java
// {RunByHand}
/****************** Exercise 27 *****************
 * Almost every Swing component is derived from
 * Component, which has a setCursor() method.
 * Look this up in the JDK documentation.
 * Create an applet and change the cursor to one
 * of the stock cursors in the Cursor class.
 ***********************************************/
import com.bruceeckel.swing.*;
import javax.swing.*;
import java.awt.*;

class Cursors extends JApplet {
  JTextField txt = new JTextField(10);
  JButton b1 = new JButton("Button 1");
  Cursor hand =
    Cursor.getPredefinedCursor(
      Cursor.HAND_CURSOR);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(txt);
    txt.setCursor(hand);
    cp.add(b1);
    setCursor(hand);
  }
}

public class E27_Cursors {
  public static void main(String args[]) {
    Console.run(new Cursors(), 200, 100);
  }
} ///:~
```

The constants in the **Cursor** class are **int**s, not **Cursor** objects, so you must use either the **Cursor** constructor or **getPredefinedCursor( )** method to generate a **Cursor** object from the **int** (this seems a bit strange, but cursors are system resources which, on Windows anyway, can be used up easily – this may be the reason for the design).

Note that if you don't also set the cursor in the **JTextField**, it goes back to the default cursor.

As an added exercise, detect every time the cursor crosses into the applet, and cycle to a new predefined cursor each time.

Exercise 28

```
//: c14:E28_ShowMethods.java
// {RunByHand}
/****************** Exercise 28 *****************
 * Starting with ShowAddListeners.java, create
 * a program with the full functionality of
 * c10:ShowMethods.java.
 ***********************************************/
import com.bruceeckel.swing.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;

class ShowMethodsClean extends JApplet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  JTextField
    name = new JTextField(25),
    searchFor = new JTextField(25);
  JTextArea results = new JTextArea(40, 65);
  JScrollPane scrollPane = new JScrollPane(results);
  class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName(nm);
      } catch(ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      m = cl.getMethods();
      ctor = cl.getConstructors();
      // Convert to an array of cleaned Strings:
      n = new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
//      for(int i = 0; i < m.length; i++) {
//        String s = m[i].toString();
//        n[i] = StripQualifiers.strip(s);
//      }
        for(int i = 0; i < ctor.length; i++)
          n[i + m.length] = ctor[i].toString();
//      for(int i = 0; i < ctor.length; i++) {
//        String s = ctor[i].toString();
//        n[i + m.length] =
//          StripQualifiers.strip(s);
//      }
      reDisplay();
    }
  }
  void reDisplay() {
    results.setText("");
    String[] rs = new String[n.length];
    if(searchFor.getText().trim().length() == 0)
      // Include everything:
      for(int i = 0; i < n.length; i++)
        results.append(n[i] + "\n");
//        results.append(
//          StripQualifiers.strip(n[i]) + "\n");
```

```
      else {
        // Include only methods that have ALL the
        // words listed in searchFor:
        StringTokenizer st =
          new StringTokenizer(searchFor.getText());
        ArrayList lookFor = new ArrayList();
        while(st.hasMoreTokens())
          lookFor.add(st.nextToken());
        for(int i = 0; i < n.length; i++) {
          Iterator it = lookFor.iterator();
          boolean include = true;
          while(it.hasNext())
            if(n[i].indexOf((String)it.next()) == -1)
              include = false;
          if(include == true)
            results.append(n[i] + "\n");
//            results.append(
//              StripQualifiers.strip(n[i]) + "\n");
        }
      }
      // Force the scrollpane back to the top:
      scrollPane.getViewport().setViewPosition(
        new Point(0, 0));
    }
    public void init() {
      name.addActionListener(new NameL());
      searchFor.addActionListener(new NameL());
      JPanel top1 = new JPanel();
      top1.add(new JLabel(
        "Qualified.class.name (press ENTER):"));
      top1.add(name);
      JPanel top2 = new JPanel();
      top2.add(new JLabel(
        "Words to search for (optional):"));
      top2.add(searchFor);
      JPanel top = new JPanel(new GridLayout(2,1));
      top.add(top1);
      top.add(top2);
      Container cp = getContentPane();
      cp.add(BorderLayout.NORTH, top);
      cp.add(scrollPane);
```

```
  }
}

public class E28_ShowMethods {
  public static void main(String args[]) {
    Console.run(new ShowMethodsClean(), 500, 400);
  }
} ///:~
```

A second **JTextField** called **searchFor** is added to hold a list of words to search for. The **NameL ActionListener** is attached to both **JTextField**s so that it updates whenever you press either one (and hit "Return"). After the **Class** object is procured and the methods and constructors are copied into **n** (the array of **String**s holding all the possible names), **reDisplay( )** is called to decide which methods to show and to display them.

If there are no words in the **searchFor** field, everything is shown (using **StripQualifiers.strip( )** from the book). If there are words in the **searchFor** field, the **String** from that field must be broken up into words; the easiest way to do this is using **StringTokenizer**, which defaults to tokenizing on white space. Each of these words is put into the **ArrayList lookFor**, and then (by using the flag **include**) each method/constructor is examined to see if it contains *all* the words in **lookFor**. If so, it is placed into the **JTextArea**.

I discovered that it was possible for the **JTextArea** to appear to be blank even though there were elements in the output, because it had scrolled off the bottom of the page. The simple solution is to move the scrollpane back to the top whenever the data in the **JTextArea** is modified. After some poking around in the Java docs and some experimentation, I discovered that I had to modify the **Viewport**, and not the **JScrollPane** directly, as you can see in the line of code at the end of **reDisplay( )**.

Exercise 29

```
//: c14:E29_TestRegularExpression2.java
// {RunByHand}
/******************** Exercise 29 ***********************
 * Turn c12:TestRegularExpression.java into an interactive
 * Swing program that allows you to put an input string in
 * one TextArea and a regular expression in a TextField.
 * The results should be displayed in a second TextArea.
 ********************************************************/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.regex.*;
import com.bruceeckel.swing.*;

public class E29_TestRegularExpression2 extends JApplet {
  private JTextArea input = new JTextArea(3, 60),
                    output = new JTextArea(3, 60);
  private JTextField expression = new JTextField(40);
  private JButton match = new JButton("Match");
  private JLabel inputL = new JLabel("Input"),
                 outputL = new JLabel("Output"),
                 expressionL = new JLabel("Expression");
  private JPanel panel1 = new JPanel(),
                 panel2 = new JPanel(),
                 panel3 = new JPanel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(3,1));
    panel1.add(inputL);
    panel1.add(new JScrollPane(input));
    cp.add(panel1);
    panel2.add(expressionL);
    panel2.add(expression);
    panel2.add(match);
    cp.add(panel2);
    panel3.add(outputL);
    panel3.add(new JScrollPane(output));
    cp.add(panel3);

    match.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String inputS = input.getText();
        String regEx = expression.getText();
        Pattern p = Pattern.compile(regEx);
        Matcher m = p.matcher(inputS);
        String outputS = "";
        while(m.find()) {
          outputS += "Match \"" + m.group() +
            "\" at positions " + m.start() + "-" +
            (m.end() - 1) + '\n';
        }
        output.setText(outputS);
      }
    });
  }
  public static void main(String[] args) {
    Console.run(
      new E29_TestRegularExpression2(), 700, 400);
  }
} ///:~
```

Exercise 30

```
//: c14:E30_InvokeLaterFrame2.java
// {RunByHand}
// Eliminating race Conditions using Swing Components.
/********************* Exercise 30 ********************
 * Modify InvokeLaterFrame.java to use invokeAndWait()
 ******************************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.Console;

public class E30_InvokeLaterFrame2 extends JFrame {
  private JTextField statusField =
    new JTextField("Initial Value");
  public E30_InvokeLaterFrame2() {
    Container cp = getContentPane();
    cp.add(statusField, BorderLayout.NORTH);
    addWindowListener(new WindowAdapter() {
      public void windowOpened(WindowEvent e) {
        try { // Simulate initialization overhead
          Thread.sleep(2000);
        } catch (InterruptedException ex) {
          throw new RuntimeException(ex);
        }
        statusField.setText("Initialization complete");
      }
    });
  }
  public static void main(String[] args) throws Exception {
    final E30_InvokeLaterFrame2 ilf =
      new E30_InvokeLaterFrame2();
    Console.run(ilf, 150, 60);
    SwingUtilities.invokeAndWait(new Runnable() {
      public void run() {
        ilf.statusField.setText("Application ready");
      }
    });
    System.out.println("Done");
  }
} ///:~
```

Chapter 15

Exercise 1

```
//: c15:E01_AssertAssertions.java
// {ThrowsException}
/****************** Exercise 1 *****************
 * Create a class containing a static clause that
 * throws an exception if assertions are not
 * enabled. Demonstrate that this test works
 * correctly.
 ***********************************************/

public class E01_AssertAssertions {
  static {
    boolean assertionsEnabled = false;
    assert assertionsEnabled = true;
    if(!assertionsEnabled)
      throw new RuntimeException("Assertions disabled");
  }
  public static void main(String[] args) {
    assert false: "Assertions Enabled";
  }
} ///:~
```

When the **boolean** value **assertionsEnabled** is set to **true,** its truth is asserted at the same time. This line is ignored if assertions are not enabled, so the **throw** clause in the static initializer will throw an exception with the message "Assertions disabled." If assertions are enabled, the assert statement in **main( )** will throw an assertion error. (This is not exactly what was asked for in the exercise description, but it's more interesting because it uses the message form of the **assert** statement).

Try running this with assertions both enabled and disabled:

```
java -ea E01_AssertAssertions
java E01_AssertAssertions
```

Exercise 2

```
//: c15:E02_LoaderAssertions2.java
/****************** Exercise 2 ******************
 * Modify the above example to use the approach in
 * LoaderAssertions.java to turn on assertions
 * instead of throwing an exception. Demonstrate
 * that this works correctly.
 ***********************************************/

public class E02_LoaderAssertions2 {
  static {
    boolean assertionsEnabled = false;
    assert assertionsEnabled = true;
    if(!assertionsEnabled)
      ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true);
  }
  public static void main(String[] args) {
    boolean assertionsEnabled2 = false;
    assert assertionsEnabled2 = true;
    if(!assertionsEnabled2)
      System.err.println("Assertions not enabled");
  }
} ///:~
```

This turns out to be a trick question, because the approach used in Exercise 1 – the static clause – won't work. If you run the program above, you'll discover that it still prints "Assertions not enabled" if you don't use the **–ea** flag. This happens because assertions have been changed in the class loader, and thus only have an effect on classes that are loaded *after* the classloader's assertion status has been changed. Here's an example that demonstrates the issue more clearly, and solves the problem (at least, as much as it can be solved):

```
//: c15:E02_LoaderAssertions3.java
// Changing the assertion status only affects classes
// loaded after it was changed.

class AssertionTest {
  private boolean assertionsEnabled = false;
  public AssertionTest() {
    assert assertionsEnabled = true;
    System.out.println(this);
  }
  public String toString() {
    return "assertionsEnabled = " + assertionsEnabled;
  }
  public boolean isOn() { return assertionsEnabled; }
}

public class E02_LoaderAssertions3 {
  public static void main(String[] args) {
    boolean enabled = false;
    assert enabled = true;
    if(!enabled) {
      System.out.println("Enabling Assertions");
      ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true);
    }
    assert enabled = true;
    System.out.println("enabled = " + enabled);
    new AssertionTest();
  }
} ///:~
```

When you run the program without the **–ea** flag, you'll get the following output:

```
Enabling Assertions
enabled = false
assertionsEnabled = true
```

Inside **main( )**, enabling assertions in the classloader has no effect. However, since **AssertionTest** is loaded after the change, it is affected. Try creating an **AssertionTest** object before the classloader is changed to see what happens.

Exercise 3

```
//: c15:E03_LoggingLevels2.java
/****************** Exercise 3 ***********************
```

```
 * In LoggingLevels.java, comment out the code that
 * sets the severity level of the root logger
 * handlers and verify that messages of level
 * CONFIG and below are not reported.
 ***********************************************/
import java.util.logging.*;

public class E03_LoggingLevels2 {
  private static Logger
    lgr = Logger.getLogger("com"),
    lgr2 = Logger.getLogger("com.bruceeckel"),
    util = Logger.getLogger("com.bruceeckel.util"),
    test = Logger.getLogger("com.bruceeckel.test"),
    rand = Logger.getLogger("random");
  private static void logMessages() {
    lgr.info("com : info");
    lgr2.info("com.bruceeckel : info");
    util.info("util : info");
    test.severe("test : severe");
    rand.info("random : info");
  }
  public static void main(String[] args) {
    // lgr.setLevel(Level.SEVERE);
    // System.out.println("com level: SEVERE");
    logMessages();
    util.setLevel(Level.FINEST);
    test.setLevel(Level.FINEST);
    rand.setLevel(Level.FINEST);
    System.out.println("individual loggers set to FINEST");
    logMessages();
    // lgr.setLevel(Level.SEVERE);
    // System.out.println("com level: SEVERE");
    // logMessages();
  }
} ///:~
```

Here's the output from one run, showing that there are no CONFIG and below messages:

```
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: com : info
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
```

```
INFO: com.bruceeckel : info
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: util : info
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
SEVERE: test : severe
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: random : info
individual loggers set to FINEST
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: com : info
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: com.bruceeckel : info
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: util : info
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
SEVERE: test : severe
Dec 2, 2003 4:02:27 PM E03_LoggingLevels2 logMessages
INFO: random : info
```

Exercise 4

```
//: c15:E04_LoggingLevels3.java
/****************** Exercise 4 ********************
 * Inherit from java.util.Logging.Level and define
 * your own level, with a value less than FINEST.
 * Modify LoggingLevels.java to use your new level
 * and show that messages at your level will not
 * appear when the logging level is FINEST.
 ***********************************************/
import java.util.logging.*;

public class E04_LoggingLevels3 {
 private static Logger lgr = Logger.getLogger("com");
  public static void main(String[] args) {
    Handler[] handlers =
      Logger.getLogger("").getHandlers();
    for(int i = 0; i < handlers.length; i++)
      handlers[i].setLevel(Level.FINEST);
    Level lessThanFinest = new Level(
      "LessThanFinest", Level.FINEST.intValue() - 1) {};
    lgr.setLevel(Level.FINEST);
    lgr.log(lessThanFinest, "Less Than Finest");
  }
} ///:~
```

An anonymous inner class simplifies the work needed to inherit from **Level**. When the logger level is set to **FINEST**, all messages at level **lessThanFinest** (**FINEST-1**) will be ignored.

Exercise 5

```
//: c15:E05_RootFileHandler.java
/****************** Exercise 5 ******************
 *  Associate a FileHandler with the root logger.
***********************************************/
import java.util.logging.*;

public class E05_RootFileHandler {
  public static void main(String[] args) throws Exception {
    Logger.getLogger("").addHandler(
      new FileHandler("E05_RootFileHandler.xml"));
    Logger.getLogger("E05_RootFileHandler").info("Hello");
  }
} ///:~
```

Note that unless the parent handlers are disabled using **setUseParentHandlers(false)**, the **LogRecord**s are also passed to those handlers. The additional **FileHandler** associated with the root handler will get the **LogRecord** from the logger named **E05_RootFileHandler**.

Exercise 6

```
//: c15:E06_RootFileHandler2.java
/****************** Exercise 6 *******************
 * Modify the FileHandler so that it formats output
 * to a simple text file.
 ***********************************************/
import java.util.logging.*;

public class E06_RootFileHandler2 {
  public static void main(String[] args) throws Exception {
    FileHandler fh =
      new FileHandler("E06_RootFileHandler2.txt");
    fh.setFormatter(new SimpleFormatter());
    Logger.getLogger("").addHandler(fh);
    Logger.getLogger("E06_RootFileHandler").info("Hello");
  }
} ///:~
```

This uses a **SimpleFormatter** object to format the output in simple text form.

Exercise 7

```
//: c15:E07_MultipleHandlers2.java
/****************** Exercise 7 ******************
 * Modify MultipleHandlers.java so that it
 * generates output in plain text format instead
 * of XML.
 ***********************************************/
import java.util.logging.*;

public class E07_MultipleHandlers2 {
  private static Logger logger =
    Logger.getLogger("E07_MultipleHandlers2");
  public static void main(String[] args) throws Exception {
    FileHandler logFile =
      new FileHandler("E07_MultipleHandlers2.txt");
    logFile.setFormatter(new SimpleFormatter());
    logger.addHandler(logFile);
    logger.addHandler(new ConsoleHandler());
    logger.warning("Output to multiple handlers");
  }
} ///:~
```

Exercise 8

```
//: c15:E08_LoggingLevels4.java
/****************** Exercise 8 ******************
 * Modify LoggingLevels.java to set different
 * logging levels for the handlers associated
 * with the root logger.
 ***********************************************/
import java.util.logging.*;

public class E08_LoggingLevels4 {
  private static Logger
    lgr = Logger.getLogger("com"),
    lgr2 = Logger.getLogger("com.bruceeckel"),
    util = Logger.getLogger("com.bruceeckel.util"),
    test = Logger.getLogger("com.bruceeckel.test"),
    rand = Logger.getLogger("random");
  private static void logMessages() {
    lgr.info("com : info");
    lgr2.info("com.bruceeckel : info");
    util.info("util : info");
    test.severe("test : severe");
    rand.info("random : info");
  }
  public static void main(String[] args) {
    Handler[] handlers =
      Logger.getLogger("").getHandlers();
    for(int i = 0; i < handlers.length; i++)
      handlers[i].setLevel(Level.SEVERE);
    lgr.setLevel(Level.SEVERE);
    System.out.println("com level: SEVERE");
    logMessages();
    util.setLevel(Level.FINEST);
    test.setLevel(Level.FINEST);
    rand.setLevel(Level.FINEST);
    System.out.println("individual loggers set to FINEST");
    logMessages();
    lgr.setLevel(Level.SEVERE);
    System.out.println("com level: SEVERE");
    logMessages();
  }
} ///:~
```

Exercise 9

```java
//: c15:E09_CustomRootLevel.java
// {Args: 299}
/****************** Exercise 9 *****************
 * Write a simple program that sets the root
 * logger logging level based on a command-line
 * argument.
 ***********************************************/
import java.util.logging.*;

public class E09_CustomRootLevel {
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(
        "Usage: java E09_CustomRootLevel intLevel");
      System.exit(1);
    }
    int level = Integer.parseInt(args[0]);
    Logger logger = Logger.getLogger("");
    Level customLevel =
      new Level("CustomLevel", level) {};
    logger.setLevel(customLevel);
    // Set all the handlers to apply the level, too:
    Handler[] handlers = logger.getHandlers();
    for(int i = 0; i < handlers.length; i++)
      handlers[i].setLevel(customLevel);
    logger.fine("Fine");
    logger.finer("Finer");
    logger.finest("Finest");
    logger.severe("Severe");
    logger.warning("Warning");
    logger.info("Info");
    logger.config("Config");
  }
} ///:~
```

This also shows the integer values corresponding to the Level constants. Experimenting with different values at the command prompt will produce different results. Note the use of the logger methods **info( )**, **fine( )**,

config( ), etc. which are convenience methods so that you don't need to pass in the logging level with each call to *log*.

Exercise 10

```
//: c15:E10_HtmlHandlerFormatter.java
/****************** Exercise 10 *****************
 * Write an example using Formatters and Handlers
 * to output a log file as HTML.
 ***********************************************/
import java.io.*;
import java.util.logging.*;
import java.util.*;

class HtmlHandler extends Handler {
  BufferedWriter out;
  public HtmlHandler(String fileName) throws IOException {
    out = new BufferedWriter(new OutputStreamWriter(
      new FileOutputStream(fileName)));
    out.write("<html><table cellpadding='1' " +
      "cellspacing='1' border='1' style='width:100%;'>");
    out.write("<h2>Start of HTML Log</h2>");
  }
  public void publish(LogRecord logRecord) {
    try {
      out.write("\n" + getFormatter().format(logRecord));
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public void flush() {
    try {
      out.flush();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public void close() {
    try {
      out.write("\n</table></html>");
      out.close();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

public class E10_HtmlHandlerFormatter {
  public static void main(String[] args)
      throws IOException {
    Logger logger =
      Logger.getLogger("E10_HtmlHandlerFormatter");
    HtmlHandler handler =
      new HtmlHandler("E10_HtmlHandlerFormatter.html");
    handler.setFormatter(new java.util.logging.Formatter(){
      Date date = new Date();
      public String format(LogRecord record) {
        date.setTime(record.getMillis());
        return "<tr><td>" + date + "</td>" +
          "<td>" + record.getSourceClassName() +  "</td>" +
          "<td>" + record.getSourceMethodName() + "</td>" +
          "<td><b>" + record.getLevel() + "</b></td>" +
          "<td>" + record.getMessage() + "</td></tr>";
      }
    });
    logger.addHandler(handler);
    logger.setUseParentHandlers(false);
    for(int i = 0; i < 10; i++) {
      logger.log(Level.WARNING, "Warning Message "+ i);
      logger.log(Level.SEVERE, "Severe Message " + i);
      logger.log(Level.INFO, "Info Message " + i);
    }
  }
} ///:~
```

An HML table seemed to be a helpful way to present the logging information.

The **Formatter** subclass is created as an anonymous inner class.

Exercise 11

```
//: c15:E11_DiffLogDest.java
/****************** Exercise 11 ****************
 * Write an example using Handlers and Filters
 * to log messages with any severity level over
 * INFO in one file and any severity level
 * including and below INFO in other file. The
 * files should be written in simple text.
 ***********************************************/
import java.io.*;
import java.util.logging.*;

public class E11_DiffLogDest {
  public static void main(String[] args)
  throws IOException {
    Logger logger = Logger.getLogger("E11_DiffLogDest");
    Handler lowerHandler = new FileHandler("lower.txt");
    Handler upperHandler = new FileHandler("upper.txt");
    logger.setLevel(Level.ALL);
    upperHandler.setFilter(new MaxMinFilter(
      Integer.MAX_VALUE,Level.INFO.intValue()+1));
    lowerHandler.setFilter(new MaxMinFilter(
      Level.INFO.intValue(),Level.ALL.intValue()));
    lowerHandler.setFormatter(new SimpleFormatter());
    upperHandler.setFormatter(new SimpleFormatter());
    logger.addHandler(lowerHandler);
    logger.addHandler(upperHandler);
    logger.severe("Message Level Severe");
    logger.info("Message Level Info");
    logger.fine("Message Level Fine");
  }
  static class MaxMinFilter implements Filter {
    int max = Integer.MAX_VALUE;
    int min = Integer.MIN_VALUE;
    public MaxMinFilter (int max, int min) {
      this.max = max;
      this.min = min;
    }
    public boolean isLoggable(LogRecord logRecord) {
      int level = logRecord.getLevel().intValue();
      return (level <= max && level >= min);
    }
  }
} ///:~
```

MaxMinFilter is a custom **Filter** that implements the atypical behavior of logging records which have levels that fall between a maximum and minimum value. This has the advantage of being generalizable to different levels.

Exercise 12

```
//: c15:E12_CustomFormatter.java
/****************** Exercise 12 ****************
 * Modify log.prop to add an additional
 * initialization class that initializes a
 * custom Formatter for the logger com.
 ***********************************************/
import java.util.logging.*;

public class E12_CustomFormatter {
  private static Logger logger = Logger.getLogger("com");
  public E12_CustomFormatter() {
    Formatter formatter = new Formatter() {
      public String format(LogRecord record) {
        return "Custom formatter: " +
          record.getLevel() + " -> " +
          record.getMessage() + "\n";
      }
    };
    ConsoleHandler cHandler = new ConsoleHandler();
    cHandler.setFormatter(formatter);
    logger.addHandler(cHandler);
    logger.setUseParentHandlers(false);
  }
} ///:~
```

```
//: c15:E12_TestLogger.java
// {JVMArgs: -Djava.util.logging.config.file=log.prop}
import java.util.logging.*;
```

```
public class E12_TestLogger {
  public static void main(String[] args) {
    Logger.getLogger("com").severe("Severe Test Message");
  }
} ///:~
```

```
//:! c15:log.prop
#### Configuration File ####
# Global Params
# Handlers installed for the root logger
handlers= java.util.logging.ConsoleHandler
java.util.logging.FileHandler
# Level for root logger - is used by any logger
# that does not have its level set
.level= FINEST
# Initialization class - the public default constructor
# of this class is called by the Logging framework
# config = ConfigureLogging
config = E12_CustomFormatter

# Configure FileHandler
# Logging file name - %u specifies unique
java.util.logging.FileHandler.pattern = java%g.log
# Write 100000 bytes before rotating this file
java.util.logging.FileHandler.limit = 100000
# Number of rotating files to be used
java.util.logging.FileHandler.count = 3
# Formatter to be used with this FileHandler
java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter

# Configure ConsoleHandler
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter

# Set Logger Levels #
com.level=SEVERE
com.bruceeckel.level = FINEST
com.bruceeckel.util.level = INFO
com.bruceeckel.test.level = FINER
random.level= SEVERE
///:~
```

Run this class using

```
java -Djava.util.logging.config.file=log.prop
E12_TestLogger
```

The **build.xml** file in this book's code distribution will be configured to do this. You'll see this output:

```
Custom formatter: SEVERE -> Severe Test Message
```

The Logger **com** is initialized to use a custom **Formatter** by creating an instance of class **E12_CustomFormatter** that creates and sets the new formatter in its constructor. The class **E12_CustomFormatter** is specified in the properties file **log.prop** using the "**config=**" statement in the program execution command.

Exercise 13

```
//: c15:E13_SimpleDebugging.java
// {ThrowsException}
// Compile with -g flag to generate debugging information.
/******************* Exercise 13 **********************
 * Run JDB on SimpleDebugging.java, but do not give the
 * command catch Exception. Show that it still catches
 * the exception.
 ********************************************************/

public class E13_SimpleDebugging {
  private static void foo1() {
    System.out.println("In foo1");
    foo2();
  }
  private static void foo2() {
    System.out.println("In foo2");
    foo3();
  }
  private static void foo3() {
    System.out.println("In foo3");
    int j = 1;
    j--;
    int i = 5 / j;
  }
  public static void main(String[] args) {
    foo1();
  }
} ///:~
```

Here is the transcript of an interactive debugging session:

```
$ jdb E13_SimpleDebugging
Initializing jdb ...
> run
run E13_SimpleDebugging
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: In foo1
In foo2
In foo3

Exception occurred: java.lang.ArithmeticException
(uncaught)"thread=main", E13_S
impleDebugging.foo3(), line=23 bci=15
23        int i = 5 / j;

main[1] print j
 j = 0
main[1] step
java.lang.ArithmeticException: / by zero at
E13_SimpleDebugging.foo3(E13_SimpleDebugging.java:23)
at E13_SimpleDebugging.foo2(E13_SimpleDebugging.java:17)
at E13_SimpleDebugging.foo1(E13_SimpleDebugging.java:13)
at E13_SimpleDebugging.main(E13_SimpleDebugging.java:26)
> Exception in thread "main"
The application exited
```

To include debug information in the compiled program, you must compile the code using the **–g** option.

Although the **catch Exception** command was not given, the debugger still stopped at the **ArithmeticException**. The command "**print j**" tells the debugger to show the field values. If you attempt to continue the program using the **step** command, it displays the exception stack trace and terminates.

Exercise 14

```
//: c15:E14_SimpleDebugging.java
// {ThrowsException}
/******************** Exercise 14 *****************
 * Add an uninitialized reference to SimpleDebugging.java
 * (you’ll have to do it in a way that the compiler
 * doesn’t catch the error!) and use JDB to track down
 * the problem.
 *****************************************************/

public class E14_SimpleDebugging {
  private static void foo1() {
    String s = null;
    System.out.println("In foo1");
    int len = s.length();
    foo2();
  }
  private static void foo2() {
    System.out.println("In foo2");
    foo3();
  }
  private static void foo3() {
    System.out.println("In foo3");
    int j = 1;
    j--;
    int i = 5 / j;
  }
  public static void main(String[] args) {
    foo1();
  }
} ///:~
```

Here is a transcript of an interactive debugging session:

```
jdb E14_SimpleDebugging
Initializing jdb ...
```

```
> run
run E14_SimpleDebugging
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: In foo1

Exception occurred: java.lang.NullPointerException
(uncaught)"thread=main", E14_SimpleDebugging.foo1(),
line=14 bci=11
14        int len = s.length();

main[1] list
10    public class E14_SimpleDebugging {
11      private static void foo1() {
12        String s = null;
13        System.out.println("In foo1");
14 =>     int len = s.length();
15        foo2();
16      }
17      private static void foo2() {
18        System.out.println("In foo2");
19        foo3();
main[1] print s
 s = null
```

As with the previous example, you must compile the code using the **–g** option to include debug information in the compiled program.

The **list** command dumps the code with an arrow indicating the current execution line. Here the execution stopped at the place where a **NullPointerException** was produced. In some cases the cause might not be immediately apparent, but there is only one object reference that could be **null** on this line, so clearly the problem is that this object is uninitialized.

Exercise 15

```
//: c15:E15_CoverageTesting.java
// {RunByHand}
// {JVMArgs: -Xrunjcov:type=M}
```

```
//****************** Exercise 15 *****************
 * Perform the experiment described in the
 * "Coverage Testing" section.
 ***********************************************/

public class E15_CoverageTesting {
  // Volatile so compiler can't make any assumptions:
  static volatile boolean flag = false;
  private static void foo1() {
    System.out.println("In foo1");
    foo2();
  }
  private static void foo2() {
    System.out.println("In foo2");
    foo3();
  }
  private static void foo3() {
    System.out.println("In foo3");
    if(false) foo4();
  }
  private static void foo4() {
    System.out.println("In foo4");
  }
  public static void main(String[] args) {
    foo1();
  }
} ///:~
```

Here is the relevant text from the book:

"To get coverage testing information for **SimpleDebugging.java**, you use the command:

```
java -Xrunjcov:type=M SimpleDebugging
```

As an experiment, try putting lines of code that will not be executed into **SimpleDebugging.java** (you'll have to be somewhat clever about this since the compiler can detect unreachable lines of code)."

Notice that this is an "X" option for java. You can list these options with the command:

```
java -X
```

At the end of the list, they note that “The -X options are non-standard and subject to change without notice.” Thus, it’s possible that if you have a JDK later than 1.4.1 (the one used for this book), it might no longer support coverage testing. In the version of the JDK that I was using, “java –X” didn’t display the **runjcov** option (even though it worked) so this command may be on its way in or out. This option wasn’t available in the JDK 1.5 beta.

First, we fixed the program so that it doesn’t throw an exception, then we added the non-executing lines of code by testing a **volatile** flag (this way, the compiler can’t make any assumptions about the state of the flag). Notice the **JVMArgs** directive which specifies the argument that you will see in the Ant **build.xml** file. So in this case, the command to run the program is:

```
java -Xrunjcov:type=M E15_CoverageTesting
```

After this command is run, you’ll find a file called **java.jcov** in the directory where the program was run. If you open this file you’ll find the coverage information, which includes every call made in the Java standard libraries as well, so you’ll have to search for **E15_CoverageTesting** to find the section of interest.

Exercise 16

```
//: c15:E16_CodingConventionDoclet.java
/****************** Exercise 16 *****************
 * Create a doclet that displays identifiers that
 * might not follow the Java naming convention by
 * checking how capital letters are used for
 * those identifiers.
 ***********************************************/
package c15;
import java.io.*;
import com.sun.javadoc.*;
import java.util.regex.*;

public class E16_CodingConventionDoclet {
```

```
public static boolean start(RootDoc root) {
  ClassDoc[] cls = root.classes();
  for(int i = 0; i < cls.length; i++) {
    // Check class name
    String klass = cls[i].name();
    System.out.println("Testing class " + klass);
    testIdentifier(klass, "[A-Z]{1}\\d*[_]?[a-zA-Z]*");
    // Check field names
    FieldDoc[] fd = cls[i].fields();
    for(int j = 0; j < fd.length; j++) {
      String fieldName = fd[j].name();
      System.out.println("Testing field " + fieldName);
      testIdentifier(fieldName, "[a-z]+[a-zA-Z]*");
    }
    // Check method names
    MethodDoc[] md = cls[i].methods();
    for(int j = 0; j < md.length; j++) {
      String methodName = md[j].name();
      System.out.println("Testing method " + methodName);
      testIdentifier(methodName, "[a-z]+[a-zA-Z]*");
    }
    // Check enclosing package name
    PackageDoc pkg = cls[i].containingPackage();
    System.out.println("Testing package " + pkg.name());
    testIdentifier(pkg.name(), "[a-z]*");
  }
  return true;
}
private static void
testIdentifier(String source, String regEx) {
  Pattern p = Pattern.compile(regEx);
  Matcher m = p.matcher(source);
  if(!m.matches())
    System.out.println("'" + source + "'" +
       " should match regular expression : " + regEx);
}
void BadMethodName() {} // Test
class badClassName {} // Test
public static void main(String args[]) throws Exception {
  Process javadocProcess = Runtime.getRuntime().exec(
    "javadoc -docletpath .. -doclet" +
        " c15.E16_CodingConventionDoclet" +
        " -private E16_CodingConventionDoclet.java");
      BufferedReader results = new BufferedReader(
        new InputStreamReader(
          javadocProcess.getInputStream()));
      String s;
      while((s = results.readLine())!= null)
      System.out.println(s);
    }
} ///:~
```

Note the use of regular expressions to solve the problem of looking for incorrect capitalization.

The **javadoc** command is executed using the **Runtime.exec( )** command in **main( )**. To see the results, the **Process** object returned by the calls to **exec( )** is captured. You can get an **InputStream** object from a **Process**, and this will provide the results that would have been output to the console.

A: Passing & Returning Objects

Exercise 1

```
//: appendixA:E01_TwoLevelAliasing.java
/****************** Exercise 1 ******************
 * Demonstrate a second level of aliasing. Create
 * a method that takes a reference to an object
 * but doesn't modify that reference's object.
 * However, the method calls a second method,
 * passing it the reference, and this second
 * method does modify the object.
 ***********************************************/
class Modifiable {
  private int i;
  Modifiable(int ii) {
    i = ii;
  }
  public void setI(int i) {
    this.i = i;
  }
  public String toString() {
    return "Modifiable i = " + i;
  }
}

class LevelOne {
  static void f(Modifiable reference) {
    LevelTwo.g(reference);
  }
}

class LevelTwo {
  static void g(Modifiable reference) {
    reference.setI(12);
  }
}

public class E01_TwoLevelAliasing {
  public static void main(String[] args) {
    Modifiable x = new Modifiable(7);
    System.out.println("x: " + x);
    System.out.println("Calling LevelOne.f(x)");
    LevelOne.f(x);
    System.out.println("x: " + x);
  }
} ///:~
```

The primary point of this exercise is to drive home the point that you're always passing references around, and thus you're always aliasing. Even if you don't directly see changes being made in the code you're writing or the method you're calling, that code/method could be calling other methods that modify the object. Here's the output:

```
x: Modifiable i = 7
Calling LevelOne.f(x)
x: Modifiable i = 12
```

Exercise 2

```
//: appendixA:E02_MyString.java
/****************** Exercise 2 ******************
 * Create a class MyString containing a String
 * object that you initialize in the constructor
 * using the constructor's argument. Add a
 * toString() method and a method concatenate()
 * that appends a String object to your internal
 * string. Implement clone() in MyString. Create
 * two static methods that each take a MyString x
 * reference as an argument and call
 * x.concatenate("test"), but in the second
 * method call clone() first. Test the two
 * methods and show the different effects.
 ***********************************************/

class MyString implements Cloneable {
  private String s;
  public MyString(String s) {
    this.s = s;
  }
  public String toString() {
    return "MyString: " + s;
  }
  public void concatenate(String add) {
    s += add;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MyObject can't clone");
    }
    return o;
  }
}

public class E02_MyString {
  public static void concat1(MyString x) {
    x.concatenate(" test");
  }
  public static void concat2(MyString x) {
    x = (MyString)x.clone();
    x.concatenate(" test");
  }
  public static void main(String args[]) {
    MyString
      x = new MyString("A Test String"),
      y = new MyString("A 2nd Test String");
    System.out.println("x = " + x);
    concat1(x);
    System.out.println("after concat1, x = " + x);
    System.out.println("y = " + y);
    concat2(y);
    System.out.println("after concat1, y = " + y);
  }
} ///:~
```

The output is:

```
x = MyString: A Test String
after concat1, x = MyString: A Test String test
y = MyString: A 2nd Test String
after concat1, y = MyString: A 2nd Test String
```

Notice that **concat1( )** modifies **x**, while **concat2( )** leaves **y** alone because it clones **y** first.

Also notice that the call to **super.clone( )** appears to take care of the copy, but remember that it's not performing a deep copy, but rather a shallow copy – it simply copies the reference to **s**, and doesn't make a duplicate of what **s** points to.

Exercise 3

```
//: appendixA:E03_Battery.java
/****************** Exercise 3 ******************
 * Create a class called Battery containing an
 * int that is a battery number (as a unique
 * identifier). Make it cloneable and give it a
 * toString() method. Now create a class called
 * Toy that contains an array of Battery and a
 * toString() that prints out all the batteries.
 * Write a clone() for Toy that automatically
 * clones all of its Battery objects. Test this
 * by cloning Toy and printing the result.
 ***********************************************/

class Battery implements Cloneable {
  static int counter = 1;
  int id = counter++;
  String s = "";
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
    ((Battery)o).s += " Cloned";
    return o;
  }
  public String toString() {
    return "Battery address: " + hashCode() +
           ", id: " + id + s;
  }
  // Test for Battery:
  public static void main(String[] args) {
    Battery
      a = new Battery(),
      b = (Battery)a.clone(),
      c = (Battery)b.clone();
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}

class Toy implements Cloneable {
  Battery[] batteries = {
    new Battery(), new Battery()
  };
  String s = "";
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e ) {
      throw new RuntimeException(e);
    }
    Toy t = (Toy)o;
    // Must clone the array of references!!:
    t.batteries = (Battery[])batteries.clone();
    for(int i = 0; i < batteries.length; i++)
      t.batteries[i] =
        (Battery)((batteries[i]).clone());
    t.s += "Cloned ";
    return o;
  }
  public String toString() {
    String s1 = s + " Toy batteries: \n";
    for(int i = 0; i < batteries.length; i++)
      s1 += "\t" + i + " " + batteries[i] + "\n";
    return s1;
  }
}

public class E03_Battery {
  public static void main(String args[]) {
    Toy t1 = new Toy(), t2 = (Toy)t1.clone();
    System.out.println("t1 = " + t1);
    System.out.println("t2 = " + t2);
  }
} ///:~
```

The real trick to this exercise is to remember that "everything is an object." And that does mean *everything*, including the array – this is especially hard to remember if you're coming from C, where arrays are primitives. You must remember to clone the array object as well. Conveniently, the built-in **clone( )** for array works just fine.

Here is the output:

```
t1 =  Toy batteries:
        0 Battery address: 4177328, id: 1
        1 Battery address: 4097744, id: 2

t2 = Cloned  Toy batteries:
        0 Battery address: 328041, id: 1 Cloned
        1 Battery address: 2083945, id: 2 Cloned
```

Exercise 4

```
//: appendixA:E04_CheckCloneable.java
```

```
/****************** Exercise 4 ******************
 * Change CheckCloneable.java so that all of the
 * clone() methods catch the
 * CloneNotSupportedException rather than passing
 * it to the caller.
 ***********************************************/
class Ordinary {}

// Overrides clone, but doesn't implement
// Cloneable:
class WrongClone extends Ordinary {
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(
        "WrongClone: Ordinary.clone() failed", e);
    }
    return o;
  }
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary
implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(
        "IsCloneable: Ordinary.clone() failed", e);
    }
    return o;
  }
}

// Turn off cloning by throwing the exception.
// However, note that the exception specification
// must be changed since the base class no longer
// throws CloneNotSupportedException, so the
// overridden method cannot either:
class NoMore extends IsCloneable {
  public Object clone() {
    return new RuntimeException(
      "Cannot clone NoMore",
      new CloneNotSupportedException());
  }
}

class TryMore extends NoMore {
  public Object clone()  {
    // No longer needs to catch the exception:
    return super.clone();
  }
}

class BackOn extends NoMore {
  private BackOn duplicate(BackOn b) {
    // Somehow make a copy of b
    // and return that copy. This is a dummy
    // copy, just to make the point:
    return new BackOn();
  }
  public Object clone() {
    // Doesn't call NoMore.clone():
    return duplicate(this);
  }
}

// Can't inherit from this, so can't override
// the clone method like in BackOn:
final class ReallyNoMore extends NoMore {}

class CheckCloneable {
  static Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    System.out.println("Attempting " + id);
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      // No longer need to catch the exception:
```

```
      x = (Ordinary)((IsCloneable)ord).clone();
      System.out.println("Cloned " + id);
    } else {
      System.out.println(
        "Not instance of Cloneable");
    }
    return x;
  }
}

public class E04_CheckCloneable {
  public static void main(String args[]) {
    // Upcasting:
    Ordinary[] ord = {
      new IsCloneable(),
      new WrongClone(),
      new NoMore(),
      new TryMore(),
      new BackOn(),
      new ReallyNoMore(),
    };
    Ordinary x = new Ordinary();
    // This won't compile, since clone() is
    // protected in Object:
    //! x = (Ordinary)x.clone();
    // tryToClone() checks first to see if
    // a class implements Cloneable:
    for(int i = 0; i < ord.length; i++)
      try {
        CheckCloneable.tryToClone(ord[i]);
      } catch (Exception e) {
        System.out.println(
          "Exception during tryToClone(): "
          + e.getMessage());
      }
  }
} ///:~
```

Initially, this looks like one of those exercises that forces you to study and understand an example from the book, which is certainly one of the goals.

But in the process of solving it, we end up learning more about exceptions.

The exceptions are handled by converting them to **RuntimeExceptions**. The **RuntimeException** constructor used here also takes a **String** argument, which is an additional message you want to include in the exception.

At first, it appears that all we have to do to meet the criteria of the exercises is to remove the exception specification from all the **clone( )** methods and replace the body with one containing a **try** block, as you can see in **WrongClone.clone( )**. This pattern is followed in **IsCloneable**, but in **NoMore** the exception specification must be removed in **clone( )** because the class is inherited from **IsCloneable**, whose **clone( )** does not throw any exceptions – remember that a derived-class method cannot throw any exceptions that aren't specified in the base-class method that it overrides. Becase the base-class method doesn't throw a checked exception, neither can the overriding method, so the only exception it *can* throw is an unchecked **RuntimeException**.

In **TryMore.clone( )**, the base-class **clone( )** method no longer throws an exception, so the overriding method cannot, and doesn't need to worry about it – it just calls **super.clone( )**.

The **BackOn** and **ReallyNoMore** classes don't change.

The **CheckCloneable.tryToClone( )** method has been modified from what's in the printed 3rd edition of the book in order to improve the output. The results are:

```
Attempting IsCloneable
Cloned IsCloneable
Attempting WrongClone
Not instance of Cloneable
Attempting NoMore
Exception during tryToClone(): Cannot clone NoMore
java.lang.CloneNotSupportedException
  at NoMore.clone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:65)
  at CheckCloneable.tryToClone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:102)
```

```
  at E04_CheckCloneable.main(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:131)

Attempting TryMore
Exception during tryToClone(): Cannot clone NoMore
java.lang.CloneNotSupportedException
  at NoMore.clone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:65)
  at TryMore.clone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:74)
  at CheckCloneable.tryToClone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:102)
  at E04_CheckCloneable.main(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:131)

Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Exception during tryToClone(): Cannot clone NoMore
java.lang.CloneNotSupportedException
  at NoMore.clone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:65)
  at CheckCloneable.tryToClone(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:102)
  at E04_CheckCloneable.main(C:/aaa-TIJ2-
solutions/code/cA/E04_CheckCloneable.java:131)
```

Exercise 5

```
//: appendixA:E05_MutableCompanion.java
/****************** Exercise 5 ******************
 * Using the mutable-companion-class technique,
 * make an immutable class containing an int, a
 * double, and an array of char.
 ***********************************************/
class Mutable {
  private int iData;
  private double dData;
  private char[] ca;
  public Mutable(int iData, double dData, char[] ca) {
    this.iData = iData;
    this.dData = dData;
    this.ca = ca;
  }
  public Mutable(int iData, double dData, String x) {
    this.iData = iData;
    this.dData = dData;
    ca = x.toCharArray();
  }
  public Mutable add(int x) {
    iData += x;
    return this;
  }
  public Mutable add(double x) {
    dData += x;
    return this;
  }
  public Mutable multiply(int x) {
    iData *= x;
    return this;
  }
  public Mutable multiply(double x) {
    dData *= x;
    return this;
  }
  public Mutable concatenate(String x) {
    ca = (new String(ca) + x).toCharArray();
    return this;
  }
  public Immutable makeImmutable() {
    return new Immutable(iData, dData, ca);
  }
  public String toString() {
    return
      "\n\tiData = " + iData +
      "\n\tdData = " + dData +
      "\n\tca = " + new String(ca);
  }
}

class Immutable {
  private int iData;
  private double dData;
  private char[] ca;
  public Immutable(int iData, double dData, char[] ca) {
    this.iData = iData;
    this.dData = dData;
    // Create a duplicate to prevent aliasing:
    this.ca = new String(ca).toCharArray();
  }
  public Immutable(int iData, double dData, String s) {
    this.iData = iData;
    this.dData = dData;
    ca = s.toCharArray();
  }
  // Primitives always make "full copies":
  public int getIData() { return iData; }
  public double getDData() { return dData; }
  // Create a duplicate to prevent aliasing:
  public char[] getCa() {
    return new String(ca).toCharArray();
  }
  public Immutable add(int x) {
    return new Immutable(iData + x, dData, ca);
  }
  public Immutable add(double x) {
    return new Immutable(iData, dData + x, ca);
  }
  public Immutable multiply(int x) {
    return new Immutable(iData * x, dData, ca);
  }
  public Immutable multiply(double x) {
    return new Immutable(iData, dData * x, ca);
  }
  public Immutable concatenate(String x) {
    return
      new Immutable(iData, dData, (new String(ca) + x));
  }
  public Mutable makeMutable() {
    return new Mutable(iData, dData, ca);
  }
  public String toString() {
    return new Mutable(iData, dData, ca).toString();
  }
  public static Immutable modify1(Immutable y) {
    Immutable val = y.add(12);
    val = val.multiply(3);
    val = val.add(11.7);
    val = val.multiply(2.9);
    val = val.concatenate(" on a pie");
    return val;
  }
  // This produces the same result:
  public static Immutable modify2(Immutable y) {
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11.7).multiply(2.9);
    m.concatenate(" on a pie");
    return m.makeImmutable();
  }
}

public class E05_MutableCompanion {
  public static void main(String args[]) {
    Immutable i2 = new Immutable(47, 99.325,
      "Green Eggs and Spam");
    Immutable r1 = Immutable.modify1(i2);
    Immutable r2 = Immutable.modify2(i2);
    System.out.println("i2 = " + i2);
    System.out.println("r1 = " + r1);
    System.out.println("r2 = " + r2);
  }
} ///:~
```

Starting with **Immutable2.java**, most of this exercise comprises (A) bookkeeping and (B) some minor extensions and creativity when deciding what to do with the new types. This is what I came up with, but yours may be different and more creative.

The output is:

```
i2 =
   iData = 47
   dData = 99.325
   ca = Green Eggs and Spam
```

```
r1 =
   iData = 177
   dData = 321.9725
   ca = Green Eggs and Spam on a pie
r2 =
   iData = 177
   dData = 321.9725
   ca = Green Eggs and Spam on a pie
```

Notice how I use the **Mutable.toString( )** method when defining the **Immutable.toString( )** method – this is an example of the "once and only once" maxim in coding. It came in handy when I started modifying the **toString( )** method, since I only had to do it in one place. Any extra overhead is probably not going to be a problem, certainly not compared with the hassle of maintaining two versions of the same code.

Exercise 6

```
//: appendixA:E06_Compete.java
/****************** Exercise 6 ******************
 * Modify Compete.java to add more member objects
 * to classes Thing2 and Thing4 and see if you
 * can determine how the timings vary with
 * complexity-whether it's a simple linear
 * relationship or if it seems more complicated.
 ***********************************************/
import java.io.*;

class MyStringB implements Cloneable, Serializable{
  private String s;
  public MyStringB(String s) {
    this.s = s;
  }
  public String toString() {
    return "MyStringB: " + s;
  }
  public void concatenate(String add) {
    s += add;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MyObject can't clone");
    }
    return o;
  }
}

class Thing1 implements Serializable {}

class Thing2 implements Serializable {
  static final int SIZE = 20;
  Thing1 o1 = new Thing1();
//MyStringB s = new MyStringB("Thing2 member");//1
/*  MyStringB[] sa = new MyStringB[SIZE]; //2
  public Thing2() {
    for(int i = 0; i < sa.length; i++)
      sa[i] = new MyStringB("Thing2 array member");
  } */
}

class Thing3 implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing3 can't clone");
    }
    return o;
  }
}

class Thing4 implements Cloneable {
  Thing3 o3 = new Thing3();
//MyStringB s = new MyStringB("Thing4 member"); // 1
/*MyStringB[] sa = new MyStringB[Thing2.SIZE];  // 2
  public Thing4() {
    for(int i = 0; i < sa.length; i++)
```

```
      sa[i] = new MyStringB("Thing4 array member");
  } */
  public Object clone() {
    Thing4 o = null;
    try {
      o = (Thing4)super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing4 can't clone");
    }
    // Clone the fields, too:
    o.o3 = (Thing3)o3.clone();
//    o.s = (MyStringB)s.clone(); //1
/*    o.sa = (MyStringB[])sa.clone();//2
    for(int i = 0; i < sa.length; i++)
      o.sa[i] = (MyStringB)sa[i].clone(); */
    return o;
  }
}

public class E06_Compete {
  static final int SIZE = 50000;
  public static void main(String[] args)
  throws Exception {
    Thing2[] a = new Thing2[SIZE];
    for(int i = 0; i < a.length; i++)
      a[i] = new Thing2();
    Thing4[] b = new Thing4[SIZE];
    for(int i = 0; i < b.length; i++)
      b[i] = new Thing4();
    long t1 = System.currentTimeMillis();
    ByteArrayOutputStream buf =
      new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    for(int i = 0; i < a.length; i++)
      o.writeObject(a[i]);
    // Now get copies:
    ObjectInputStream in = new ObjectInputStream(
      new ByteArrayInputStream(buf.toByteArray()));
    Thing2[] c = new Thing2[SIZE];
    for(int i = 0; i < c.length; i++)
      c[i] = (Thing2)in.readObject();
    long t2 = System.currentTimeMillis();
    System.out.println("Duplication via serialization: " +
      (t2 - t1) + " Milliseconds");
    // Now try cloning:
    t1 = System.currentTimeMillis();
    Thing4[] d = new Thing4[SIZE];
    for(int i = 0; i < d.length; i++)
      d[i] = (Thing4)b[i].clone();
    t2 = System.currentTimeMillis();
    System.out.println("Duplication via cloning: " +
      (t2 - t1) + " Milliseconds");
  }
} ///:~
```

Because I now have a faster machine (and possibly a faster JVM) than the last time I ran the original program, I needed to bump the number of objects up to 50,000 instead of 5,000 so that the "duplication by cloning" time is greater than zero.

Without adding any objects to **Thing2** and **Thing4**, the output is:

```
Duplication via serialization: 1540 Milliseconds
Duplication via cloning: 110 Milliseconds
```

Adding a single **MyStringB** object to both **Thing2** and **Thing4** (by uncommenting the lines marked '**//1**'), the output is:

```
Duplication via serialization: 2250 Milliseconds
Duplication via cloning: 500 Milliseconds
```

Adding an array of **MyStringB** objects to both (by uncommenting the blocks marked '**//2**'), with a **Thing2.SIZE** of 5, we get:

```
Duplication via serialization: 7360 Milliseconds
Duplication via cloning: 390 Milliseconds
```

With a **Thing2.SIZE** of 20, I get an **OutOfMemoryError**. To increase the maximum heap size, with JDK 1.3 and 1.4 you can use the special extension (which they clearly specify may not be available in future versions of Java) **–Xmx** like this:

```
java -Xmx100m E06_Compete
```

This increases the maximum heap space from the default of 64 Mb to 100 Mb. The serialization approach still runs out of memory (after a long time). But if the serialization section is commented out, we see:

```
Duplication via cloning: 2800 Milliseconds
```

This still required the extra memory.

It's probably safe to say that the relationship "seems more complicated." And it's also useful to observe that not only does serialization consistently take a lot longer, it may also end up using significantly more memory (although further experiments would be necessary to make sure of this).

Exercise 7

```
//: appendixA:E07_DeepSnake.java
/****************** Exercise 7 ******************
 * Starting with Snake.java, create a deep-copy
 * version of the snake.
 ***********************************************/

class Snake implements Cloneable {
  private Snake next;
  private char c;
  // Value of i == number of segments
  Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  void increment() {
    c++;
    if(next != null)
      next.increment();
  }
  public String toString() {
    String s = ":" + c;
    if(next != null)
      s += next.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Snake can't clone");
    }
    Snake s = (Snake)o;
    if(s.next != null)
      s.next = (Snake)s.next.clone();
    return s;
  }
}

public class E07_DeepSnake {
  public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println(
      "after s.increment, s = " + s);
    System.out.println(
      "after s.increment, s2 = " + s2);
  }
} ///:~
```

The trick is to call **clone( )** recursively for each segment of the **Snake**. The output is:

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s = :b:c:d:e:f
after s.increment, s2 = :a:b:c:d:e
```

So you can see that incrementing **s** doesn't affect **s2**, as it did in the original shallow-copy version of the example in the book.

Exercise 8

```
//: appendixA:E08_CloningCollection.java
/****************** Exercise 8 *****************
 * Implement the Collection interface in a class
 * called CloningCollection, using a private
 * ArrayList to provide the container
 * functionality. Override the clone() method so
 * that CloningCollection performs a "conditional
 * deep copy:" it attempts to clone() all the
 * elements it contains, but if it cannot it
 * leaves the reference(s) aliased.
 ***********************************************/
import java.util.*;

class CloningCollection
  extends AbstractCollection implements Cloneable {
  private ArrayList list;
  public CloningCollection() {
    list = new ArrayList();
  }
  public CloningCollection(Collection c) {
    list = new ArrayList(c);
  }
  public boolean isEmpty() {
    return list.isEmpty();
  }
  public int hashCode() {
    return list.hashCode();
  }
  public boolean contains(Object o) {
    return list.contains(o);
  }
  public boolean equals(Object obj) {
    return list.equals(obj);
  }
  public Object clone() {
    CloningCollection cc = null;
    try {
      cc = (CloningCollection)super.clone();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
    for(int i = 0; i < list.size(); i++) {
      Object element = cc.list.get(i);
      try {
        // You can't do this because
        // Object.clone() is protected:
        // element = element.clone();
        // You can do this instead, as long as
        // the actual method is public:
        element = element.getClass().getMethod(
          "clone", null).invoke(element, null);
        System.out.println("cloned " + element);
        cc.list.set(i, element);
      } catch (Exception e) {
        System.out.println(element + " not Cloneable");
      }
    }
    return cc;
  }
  public Object[] toArray() { return list.toArray(); }
  public Object[] toArray(Object a[]) {
    return list.toArray(a);
  }
  public boolean add(Object o) { return list.add(o); }
  public boolean remove(Object o) {
    return list.remove(o);
  }
  public boolean containsAll(Collection c) {
    return list.containsAll(c);
  }
  public boolean addAll(Collection c) {
    return list.addAll(c);
  }
  public boolean removeAll(Collection c) {
    return list.removeAll(c);
  }
  public boolean retainAll(Collection c) {
    return list.retainAll(c);
  }
  public void clear() { list.clear(); }
  public String toString() { return list.toString(); }
  public Iterator iterator() { return list.iterator(); }
  public int size() { return list.size(); }
}

public class E08_CloningCollection {
  public static void main(String args[])
    throws Exception {
    CloningCollection cc = new CloningCollection(
      Arrays.asList(new Object[]{
        new MyString("Huey"),
        new MyString("Dewey"),
        new MyString("Louie")}));
    CloningCollection cc2 = (CloningCollection)cc.clone();
    System.out.println("cc2 = " + cc2);
    cc = new CloningCollection(Arrays.asList(new Object[]{
      "Curly", "Larry", "Moe"}));
    cc2 = (CloningCollection)cc.clone();
    System.out.println("cc2 = " + cc2);
    cc = new CloningCollection(
      Arrays.asList(new Object[]{
        new Integer(7),
        new Integer(11),
        new Integer(19)}));
    cc2 = (CloningCollection)cc.clone();
    System.out.println("cc2 = " + cc2);
  }
} ///:~
```

Following the instructions, we use delegation to an **ArrayList**, and implement the **clone( )** method. Your natural impulse will be to simply call **Object.clone( )** for every object in the **ArrayList**, like this:

```
element = element.clone();
```

But that won't work, because **Object.clone( )** is protected, so you get a compile-time error. Here is a situation where compile-time checking is too strict for our needs – but how do we loosen it?

The trick is to use reflection. When you use reflection, the compiler cannot know what method is actually getting called, so it can't prevent you from calling it. We can generally assume that if there's some reason you can't call that method (which will result in an exception being thrown) then that object doesn't implement cloneability properly, so we just ignore it.

In **main( )**, we try cloning a **CloningCollection** of **MyString**, which we know is **Cloneable**, and also built-in **String** and **Integer** objects, which turn out not to be **Cloneable** (as are most classes that come with Java – what does that suggest?).

The output is:

```
cloned MyString: Huey
cloned MyString: Dewey
cloned MyString: Louie
cc2 = [MyString: Huey, MyString: Dewey, MyString: Louie]
Curly not Cloneable
Larry not Cloneable
Moe not Cloneable
cc2 = [Curly, Larry, Moe]
7 not Cloneable
11 not Cloneable
19 not Cloneable
cc2 = [7, 11, 19]
```

Additional Exercise: Does delegation make sense in this case? Re-implement **CloningCollection** using inheritance.

Additional Exercise: (Challenging) Re-implement **CloningCollection** using the *dynamic proxy* class **java.lang.reflect.Proxy**.

Supplemental code

This section contains supplemental code from “Thinking in Java, 3rd edition,” used for this solution guide. The code is included here and in the code distribution for this book, so that the compilation works properly.

```java
//: com:bruceeckel:tools:CheckVersion.java
// {RunByHand}
package com.bruceeckel.tools;

public class CheckVersion {
  public static void main(String[] args) {
    String version = System.getProperty("java.version");
    char minor = version.charAt(2);
    char point = version.charAt(4);
    System.out.println("JDK version "+ version + " found");
    if(minor < '4' || (minor == '4' && point < '1'))
      throw new RuntimeException("JDK 1.4.1 or higher " +
        "is required to run the examples in this book.");
  }
} ///:~
```

```java
//: c07:Music3.java
// An extensible program.

class Instrument {
  void play(Note n) {
    System.out.println("Instrument.play() " + n);
  }
  String what() { return "Instrument"; }
  void adjust() {}
}

class Wind extends Instrument {
  void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
  String what() { return "Wind"; }
  void adjust() {}
}

class Percussion extends Instrument {
  void play(Note n) {
    System.out.println("Percussion.play() " + n);
  }
  String what() { return "Percussion"; }
  void adjust() {}
}

class Stringed extends Instrument {
  void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
  String what() { return "Stringed"; }
  void adjust() {}
}

class Brass extends Wind {
  void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
  void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  void play(Note n) {
    System.out.println("Woodwind.play() " + n);
  }
  String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
} ///:~
```

```
//: c07:Note.java
// Notes to play on musical instruments.

public class Note {
  private String noteName;
  private Note(String noteName) {
    this.noteName = noteName;
  }
  public String toString() { return noteName; }
  public static final Note
    MIDDLE_C = new Note("Middle C"),
    C_SHARP  = new Note("C Sharp"),
    B_FLAT   = new Note("B Flat");
    // Etc.
} ///:~
```

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator { Object next(); } ///:~
```

```
//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator { boolean next(); } ///:~
```

```
//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator { byte next(); } ///:~
```

```
//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator { char next(); } ///:~
```

```
//: com:bruceeckel:util:ShortGenerator.java
package com.bruceeckel.util;
public interface ShortGenerator { short next(); } ///:~
```

```
//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
public interface IntGenerator { int next(); } ///:~
```

```
//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator { long next(); } ///:~
```

```
//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator { float next(); } ///:~
```

```
//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator { double next(); } ///:~
```

```
//: com:bruceeckel:util:Arrays2.java
// A supplement to java.util.Arrays, to provide additional
// useful functionality when working with arrays. Allows
// any array to be converted to a String, and to be filled
// via a user-defined "generator" object.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  public static String toString(boolean[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(byte[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(char[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(short[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(int[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(long[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(float[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(double[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  // Fill an array using a generator:
  public static void fill(Object[] a, Generator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(Object[] a, int from, int to, Generator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(boolean[] a, BooleanGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(boolean[] a, int from, int to,BooleanGenerator gen)
{
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(byte[] a, ByteGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(byte[] a, int from, int to, ByteGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(char[] a, CharGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(char[] a, int from, int to, CharGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(short[] a, ShortGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(short[] a, int from, int to, ShortGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(int[] a, IntGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(int[] a, int from, int to, IntGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(long[] a, LongGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(long[] a, int from, int to, LongGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(float[] a, FloatGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(float[] a, int from, int to, FloatGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(double[] a, DoubleGenerator gen)
{
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(double[] a, int from, int to, DoubleGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  private static Random r = new Random();
  public static class
  RandBooleanGenerator implements BooleanGenerator {
    public boolean next() { return r.nextBoolean(); }
  }
  public static class
  RandByteGenerator implements ByteGenerator {
    public byte next() { return (byte)r.nextInt(); }
  }
  private static String ssource =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  private static char[] src = ssource.toCharArray();
  public static class
  RandCharGenerator implements CharGenerator {
    public char next() {
      return src[r.nextInt(src.length)];
    }
  }
  public static class
  RandStringGenerator implements Generator {
    private int len;
    private RandCharGenerator cg = new RandCharGenerator();
    public RandStringGenerator(int length) {
      len = length;
    }
    public Object next() {
      char[] buf = new char[len];
      for(int i = 0; i < len; i++)
        buf[i] = cg.next();
      return new String(buf);
    }
  }
  public static class
  RandShortGenerator implements ShortGenerator {
    public short next() { return (short)r.nextInt(); }
  }
  public static class
  RandIntGenerator implements IntGenerator {
    private int mod = 10000;
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) { mod = modulo; }
    public int next() { return r.nextInt(mod); }
  }
  public static class
  RandLongGenerator implements LongGenerator {
    public long next() { return r.nextLong(); }
  }
  public static class
  RandFloatGenerator implements FloatGenerator {
    public float next() { return r.nextFloat(); }
  }
  public static class
  RandDoubleGenerator implements DoubleGenerator {
    public double next() {return r.nextDouble();}
  }
} ///:~
```

```
//: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;

public class Pair {
  public Object key, value;
  public Pair(Object k, Object v) {
    key = k;
    value = v;
  }
} ///:~
```

```
//: com:bruceeckel:util:MapGenerator.java
package com.bruceeckel.util;
public interface MapGenerator { Pair next(); } ///:~
```

```
//: com:bruceeckel:util:Collections2.java
// To fill any type of container using a generator object.
package com.bruceeckel.util;
import java.util.*;

public class Collections2 {
  // Fill an array using a generator:
  public static void
  fill(Collection c, Generator gen, int count) {
    for(int i = 0; i < count; i++)
      c.add(gen.next());
  }
  public static void
  fill(Map m, MapGenerator gen, int count) {
    for(int i = 0; i < count; i++) {
      Pair p = gen.next();
      m.put(p.key, p.value);
    }
  }
  public static class
  RandStringPairGenerator implements MapGenerator {
    private Arrays2.RandStringGenerator gen;
    public RandStringPairGenerator(int len) {
      gen = new Arrays2.RandStringGenerator(len);
    }
    public Pair next() {
      return new Pair(gen.next(), gen.next());
    }
  }
  // Default object so you don't have to create your own:
  public static RandStringPairGenerator rsp =
    new RandStringPairGenerator(10);
  public static class
  StringPairGenerator implements MapGenerator {
    private int index = -1;
    private String[][] d;
    public StringPairGenerator(String[][] data) {
      d = data;
    }
    public Pair next() {
      // Force the index to wrap:
      index = (index + 1) % d.length;
      return new Pair(d[index][0], d[index][1]);
    }
    public StringPairGenerator reset() {
      index = -1;
      return this;
    }
  }
  // Use a predefined dataset:
  public static StringPairGenerator geography =
    new StringPairGenerator(CountryCapitals.pairs);
  // Produce a sequence from a 2D array:
  public static class StringGenerator implements Generator{
    private String[][] d;
    private int position;
    private int index = -1;
    public StringGenerator(String[][] data, int pos) {
      d = data;
      position = pos;
    }
    public Object next() {
      // Force the index to wrap:
      index = (index + 1) % d.length;
      return d[index][position];
    }
    public StringGenerator reset() {
      index = -1;
      return this;
    }
  }
  // Use a predefined dataset:
  public static StringGenerator countries =
    new StringGenerator(CountryCapitals.pairs, 0);
  public static StringGenerator capitals =
    new StringGenerator(CountryCapitals.pairs, 1);
} ///:~
```

```
//: com:bruceeckel:util:CountryCapitals.java
package com.bruceeckel.util;

public class CountryCapitals {
  public static final String[][] pairs = {
    // Africa
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"},
    {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"GUINEA","-"}, {"BISSAU","Bissau"},
    {"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
    {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
    {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
    {"RWANDA","Kigali"},
    {"SAO TOME E PRINCIPE","Sao Tome"},
    {"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
    {"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
    {"SOUTH AFRICA","Pretoria/Cape Town"},
    {"SUDAN","Khartoum"},
    {"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
    {"TOGO","Lome"}, {"TUNISIA","Tunis"},
    {"UGANDA","Kampala"},
    {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
     "Kinshasa"},
    {"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
    // Asia
    {"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
    {"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
    {"BRUNEI","Bandar Seri Begawan"},
    {"CAMBODIA","Phnom Penh"},
    {"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
    {"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
    {"IRAN","Tehran"}, {"IRAQ","Baghdad"},
    {"ISRAEL","Tel Aviv"}, {"JAPAN","Tokyo"},
    {"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
    {"LAOS","Vientiane"}, {"LEBANON","Beirut"},
    {"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
    {"MONGOLIA","Ulan Bator"},
    {"MYANMAR (BURMA)","Rangoon"},
    {"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
    {"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
    {"PHILIPPINES","Manila"}, {"QATAR","Doha"},
    {"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
    {"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
    {"SYRIA","Damascus"},
    {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
    {"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
    {"UNITED ARAB EMIRATES","Abu Dhabi"},
    {"VIETNAM","Hanoi"}, {"YEMEN","Sana'a"},
    // Australia and Oceania
    {"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
    {"KIRIBATI","Bairiki"},
    {"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
    {"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
    {"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
    {"PAPUA NEW GUINEA","Port Moresby"},
    {"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
    {"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
    {"WESTERN SAMOA","Apia"},
    // Eastern Europe and former USSR
    {"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
    {"BELARUS (BYELORUSSIA)","Minsk"},
    {"GEORGIA","Tbilisi"},
    {"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
    {"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
    {"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
    {"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
    // Europe
```

```
      {"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
      {"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
      {"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
      {"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
      {"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
      {"FINLAND","Helsinki"}, {"FRANCE","Paris"},
      {"GERMANY","Berlin"}, {"GREECE","Athens"},
      {"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
      {"IRELAND","Dublin"}, {"ITALY","Rome"},
      {"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
      {"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
      {"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
      {"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
      {"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
      {"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
      {"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
      {"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
      {"SLOVENIA","Ljujiana"}, {"SPAIN","Madrid"},
      {"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
      {"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
      // North and Central America
      {"ANTIGUA AND BARBUDA","Saint John's"},
      {"BAHAMAS","Nassau"},
      {"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
      {"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
      {"CUBA","Havana"}, {"DOMINICA","Roseau"},
      {"DOMINICAN REPUBLIC","Santo Domingo"},
      {"EL SALVADOR","San Salvador"},
      {"GRENADA","Saint George's"},
      {"GUATEMALA","Guatemala City"},
      {"HAITI","Port-au-Prince"},
      {"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
      {"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
      {"PANAMA","Panama City"}, {"ST. KITTS","-"},
      {"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
      {"ST. VINCENT AND THE GRENADINES","Kingstown"},
      {"UNITED STATES OF AMERICA","Washington, D.C."},
      // South America
      {"ARGENTINA","Buenos Aires"},
      {"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
      {"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
      {"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
      {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
      {"PERU","Lima"}, {"SURINAME","Paramaribo"},
      {"TRINIDAD AND TOBAGO","Port of Spain"},
      {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
    };
  } ///:~
```

```
//: com:bruceeckel:util:AlphabeticComparator.java
// Keeping upper and lowercase letters together.
package com.bruceeckel.util;
import java.util.*;

public class AlphabeticComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.toLowerCase().compareTo(s2.toLowerCase());
  }
} ///:~
```

```
//: com:bruceeckel:util:TextFile.java
// Static methods for reading and writing text files as
// a single string, and treating a file as an ArrayList.
// {Clean: test.txt test2.txt}
package com.bruceeckel.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList {
  // Tools to read and write files as single strings:
  public static String
  read(String fileName) throws IOException {
    StringBuffer sb = new StringBuffer();
    BufferedReader in =
      new BufferedReader(new FileReader(fileName));
    String s;
    while((s = in.readLine()) != null) {
      sb.append(s);
      sb.append("\n");
    }
    in.close();
    return sb.toString();
  }
  public static void
  write(String fileName, String text) throws IOException {
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter(fileName)));
    out.print(text);
    out.close();
  }
  public TextFile(String fileName) throws IOException {
    super(Arrays.asList(read(fileName).split("\n")));
  }
  public void write(String fileName) throws IOException {
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter(fileName)));
    for(int i = 0; i < size(); i++)
      out.println(get(i));
    out.close();
  }
  // Simple test:
  public static void main(String[] args) throws Exception {
    String file = read("TextFile.java");
    write("test.txt", file);
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
  }
} ///:~
```

```
//: c13:Timeout.java
// Set a time limit on the execution of a program
import java.util.*;

public class Timeout extends Timer {
  public Timeout(int delay, final String msg) {
    super(true); // Daemon thread
    schedule(new TimerTask() {
      public void run() {
        System.out.println(msg);
        System.exit(0);
      }
    }, delay);
  }
} ///:~
```

```
//: com:bruceeckel:swing:Console.java
// Tool for running Swing demos from the
// console, both applets and JFrames.
package com.bruceeckel.swing;
import javax.swing.*;

public class Console {
  // Create a title string from the class name:
  public static String title(Object o) {
    String t = o.getClass().toString();
    // Remove the word "class":
    if(t.indexOf("class") != -1)
      t = t.substring(6);
    return t;
  }
  public static void
  run(JFrame frame, int width, int height) {
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
  public static void
  run(JApplet applet, int width, int height) {
    JFrame frame = new JFrame(title(applet));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(applet);
    frame.setSize(width, height);
    applet.init();
    applet.start();
    frame.setVisible(true);
  }
  public static void
  run(JPanel panel, int width, int height) {
    JFrame frame = new JFrame(title(panel));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(panel);
    frame.setSize(width, height);
    frame.setVisible(true);
    }
} ///:~
```