PREV
Chapter 17: Introducing JavaFX

NEXT
ppendix B: Using Java's Documentation Comments

Appendix A

Answers to Self Tests

Chapter 1: Java Fundamentals

1 . What is bytecode and why is it important to Java's use for Internet programming?

Bytecode is a highly optimized set of instructions that is executed by the Java Virtual Machine. Bytecode helps Java achieve both portability and security.

2 . What are the three main principles of object-oriented programming?

Encapsulation, polymorphism, and inheritance.

3 . Where do Java programs begin execution?

Java programs begin execution at **main( )**.

4 . What is a variable?

A variable is a named memory location. The contents of a variable can be changed during the execution of a program.

5 . Which of the following variable names is invalid?

The invalid variable is **D**. Variable names cannot begin with a digit.

6 . How do you create a single-line comment? How do you create a multiline comment?

A single-line comment begins with // and ends at the end of the line. A multiline comment begins with /* and ends with */.

7 . Show the general form of the **if** statement. Show the general form of the **for** loop.

The general form of the **if**:

if*(condition) statement;*

The general form of the **for**:

for*(initialization; condition; iteration) statement;*

8 . How do you create a block of code?

A block of code is started with a **{** and ended with a **}**.

9 . The moon's gravity is about 17 percent that of the earth's. Write a program that computes your effective weight on the moon.

```
/*
   Compute your weight on the moon.

   Call this file Moon.java.
*/
class Moon {
  public static void main(String args[]) {
    double earthweight; // weight on earth
    double moonweight; // weight on moon
    earthweight = 165;

    moonweight = earthweight * 0.17;

    System.out.println(earthweight +
                       " earth-pounds is equivalent to " +
                       moonweight + " moon-pounds.");

  }
}
```

10. Adapt Try This 1-2 so that it prints a conversion table of inches to meters. Display 12 feet of conversions, inch by inch. Output a blank line every 12 inches. (One meter equals approximately 39.37 inches.)

```
/*
   This program displays a conversion
   table of inches to meters.

   Call this program InchToMeterTable.java.
*/
class InchToMeterTable {
  public static void main(String args[]) {
    double inches, meters;
    int counter;

    counter = 0;
    for(inches = 1; inches <= 144; inches++) {
      meters = inches / 39.37; // convert to meters
      System.out.println(inches + " inches is " +
                         meters + " meters.");

      counter++;
      // every 12th line, print a blank line
      if(counter == 12) {
        System.out.println();
        counter = 0; // reset the line counter
      }
    }
  }
}
```

11. If you make a typing mistake when entering your program, what sort of error will result?

A syntax error.

12. Does it matter where on a line you put a statement?

No, Java is a free-form language.

Chapter 2: Introducing Data Types and Operators

1. Why does Java strictly specify the range and behavior of its primitive types?

Java strictly specifies the range and behavior of its primitive types to ensure portability across platforms.

2. What is Java's character type, and how does it differ from the character type used by some other programming languages?

Java's character type is **char**. Java characters are Unicode rather than ASCII, which is used by some other computer languages.

3. A **boolean** value can have any value you like because any non-zero value is true. True or False?

False. A boolean value must be either **true** or **false**.

4. Given this output,

```
One
Two
Three
```

use a single string to show the **println( )** statement that produced it.

```
System.out.println("One\nTwo\nThree");
```

5. What is wrong with this fragment?

```
for(i = 0; i < 10; i++) {
  int sum;

  sum = sum + i;
}
System.out.println("Sum is: " + sum);
```

There are two fundamental flaws in the fragment. First, **sum** is created

will not be known outside of the block in which it is declared. Thus, the reference to it in the **println( )** statement is invalid.

6 . Explain the difference between the prefix and postfix forms of the increment operator.

When the increment operator precedes its operand, Java will perform the increment prior to obtaining the operand's value for use by the rest of the expression. If the operator follows its operand, then Java will obtain the operand's value before incrementing.

7 . Show how a short-circuit AND can be used to prevent a divide-by-zero error.

```
if((b != 0) && (val / b)) ...
```

8 . In an expression, what type are **byte** and **short** promoted to?

In an expression, **byte** and **short** are promoted to **int**.

9 . In general, when is a cast needed?

A cast is needed when converting between incompatible types or when a narrowing conversion is occurring.

10. Write a program that finds all of the prime numbers between 2 and 100.

```
// Find prime numbers between 2 and 100.
class Prime {
  public static void main(String args[]) {
    int i, j;
    boolean isprime;

    for(i=2; i < 100; i++) {
      isprime = true;

      // see if the number is evenly divisible
      for(j=2; j <= i/j; j++)
        // if it is, then it's not prime
        if((i%j) == 0) isprime = false;

      if(isprime)
        System.out.println(i + " is prime.");
    }
  }
}
```

11. Does the use of redundant parentheses affect program performance?

No.

12. Does a block define a scope?

Yes.

Chapter 3: Program Control Statements

1 . Write a program that reads characters from the keyboard until a period is received. Have the program count the number of spaces. Report the total at the end of the program.

```
// Count spaces.
class Spaces {
  public static void main(String args[])
    throws java.io.IOException {

    char ch;
    int spaces = 0;


    System.out.println("Enter a period to stop.");
    do {
      ch = (char) System.in.read();
      if(ch == ' ') spaces++;
    } while(ch != '.');

    System.out.println("Spaces: " + spaces);
  }
}
```

2 . Show the general form of the **if-else-if** ladder.

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

3 . Given

```
if(x < 10)
  if(y > 100) {
    if(!done) x = z;
    else y = z;
  }
else System.out.println("error"); // what if?
```

to what **if** does the last **else** associate?

The last **else** associates with **if(y > 100)**.

4 . Show the **for** statement for a loop that counts from 1000 to 0 by –2.

```
for(int i = 1000; i >= 0; i -= 2) // ...
```

5 . Is the following fragment valid?

```
for(int i = 0; i < num; i++)
  sum += i;

count = i;
```

No; **i** is not known outside of the **for** loop in which it is declared.

6 . Explain what **break** does. Be sure to explain both of its forms.

A **break** without a label causes termination of its immediately enclosing loop or **switch** statement. A **break** with a label causes control to transfer to the end of the labeled block.

7 . In the following fragment, after the **break** statement executes, what is displayed?

After **break** executes, “after while” is displayed.

8 . What does the following fragment print?

```
for(int i = 0; i<10; i++) {
  System.out.print(i + " ");
  if((i%2) == 0) continue;
  System.out.println();
}
```

Here is the answer:

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

9 . The iteration expression in a for loop need not always alter the loop control variable by a fixed amount. Instead, the loop control variable can change in any arbitrary way. Using this concept, write a program that uses a **for** loop to generate and display the progression 1, 2, 4, 8, 16, 32, and so on.

```
/* Use a for loop to generate the progression

   1 2 4 8 16, ...
*/
class Progress {
  public static void main(String args[]) {

    for(int i = 1; i < 100; i += i)
      System.out.print(i + " ");

  }
}
```

10. The ASCII lowercase letters are separated from the uppercase letters by 32. Thus, to convert a lowercase letter to uppercase, subtract 32 from it. Use this information to write a program that reads characters from the keyboard. Have it convert all lowercase letters to uppercase, and all uppercase letters to lowercase, displaying the result. Make no changes to any other character. Have the program stop when the user enters a period. At the end, have the program display the number of case changes that have taken place.

```
// Change case.
class CaseChg {
  public static void main(String args[])
    throws java.io.IOException {
    char ch;
    int changes = 0;

    System.out.println("Enter period to stop.");

    do {
      ch = (char) System.in.read();
      if(ch >= 'a' & ch <= 'z') {
        ch -= 32;
        changes++;
        System.out.println(ch);
      }
      else if(ch >= 'A' & ch <= 'Z') {
        ch += 32;
        changes++;
        System.out.println(ch);
      }
    } while(ch != '.');
    System.out.println("Case changes: " + changes);
  }
}
```

11. What is an infinite loop?

An infinite loop is a loop that runs indefinitely.

12. When using **break** with a label, must the label be on a block that contains the **break**?

Yes.

Chapter 4: Introducing Classes, Objects, and Methods

1 . What is the difference between a class and an object?

A class is a logical abstraction that describes the form and behavior of an object. An object is a physical instance of the class.

2 . How is a class defined?

A class is defined by using the keyword **class**. Inside the **class** statement, you specify the code and data that comprise the class.

3 . What does each object have its own copy of?

Each object of a class has its own copy of the class' instance variables.

4 . Using two separate statements, show how to declare an object called **counter** of a class called **MyCounter**.

```
MyCounter counter;
counter = new MyCounter();
```

5 . Show how a method called **myMeth( )** is declared if it has a return type of **double** and has two **int** parameters called **a** and **b**.

```
double myMeth(int a, int b) { // ...
```

6 . How must a method return if it returns a value?

A method that returns a value must return via the **return** statement, passing back the return value in the process.

7 . What name does a constructor have?

A constructor has the same name as its class.

8 . What does **new** do?

9 . What is garbage collection and how does it work? What is **finalize( )**?

Garbage collection is the mechanism that recycles unused objects so that their memory can be reused. An object's **finalize( )** method is called just prior to an object being recycled.

10. What is **this**?

The **this** keyword is a reference to the object on which a method is invoked. It is automatically passed to a method.

11. Can a constructor have one or more parameters?

Yes.

12. If a method returns no value, what must its return type be?

void

Chapter 5: More Data Types and Operators

1 . Show two ways to declare a one-dimensional array of 12 **double**s.

```
double x[] = new double[12];
double[] x = new double[12];
```

2 . Show how to initialize a one-dimensional array of integers to the values 1 through 5.

```
int x[] = { 1, 2, 3, 4, 5 };
```

3 . Write a program that uses an array to find the average of ten **double** values. Use any ten values you like.

```
// Average 10 double values.
class Avg {
  public static void main(String args[]) {
    double nums[] = { 1.1, 2.2, 3.3, 4.4, 5.5,
                      6.6, 7.7, 8.8, 9.9, 10.1 };
    double sum = 0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i];

    System.out.println("Average: " + sum / nums.length);
  }
}
```

4 . Change the sort in Try This 5-1 so that it sorts an array of strings. Demonstrate that it works.

```
// Demonstrate the Bubble sort with strings.
class StrBubble {
  public static void main(String args[]) {
    String strs[] = {
                     "this", "is", "a", "test",
                     "of", "a", "string", "sort"
                    };
    int a, b;
    String t;
    int size;

    size = strs.length; // number of elements to sort

    // display original array
    System.out.print("Original array is:");
    for(int i=0; i < size; i++)
      System.out.print(" " + strs[i]);
    System.out.println();

    // This is the bubble sort for strings.
    for(a=1; a < size; a++)
      for(b=size-1; b >= a; b--) {
        if(strs[b-1].compareTo(strs[b]) > 0) { // if out of order
          // exchange elements
          t = strs[b-1];
          strs[b-1] = strs[b];
          strs[b] = t;
        }
      }
    // display sorted array
    System.out.print("Sorted array is:");
    for(int i=0; i < size; i++)
      System.out.print(" " + strs[i]);
    System.out.println();
  }
}
```

5 . What is the difference between the **String** methods **indexOf( )** and **lastIndexOf( )**?

The **indexOf( )** method finds the first occurrence of the specified substring. **lastIndexOf( )** finds the last occurrence.

6 . Since all strings are objects of type **String**, show how you can call the **length( )** and **charAt( )** methods on this string literal: "I like Java".

As strange as it may look, this is a valid call to **length( )**:

```
System.out.println("I like Java".length());
```

The output displayed is 11. **charAt( )** is called in a similar fashion.

```
// An improved XOR cipher.
class Encode {
  public static void main(String args[]) {
    String msg = "This is a test";
    String encmsg = "";
    String decmsg = "";
    String key = "abcdefgi";
    int j;

    System.out.print("Original message: ");
    System.out.println(msg);

    // encode the message
    j = 0;
    for(int i=0; i < msg.length(); i++) {
      encmsg = encmsg + (char) (msg.charAt(i) ^ key.charAt(j));
      j++;
      if(j==8) j = 0;
    }

    System.out.print("Encoded message: ");
    System.out.println(encmsg);

    // decode the message
    j = 0;
    for(int i=0; i < msg.length(); i++) {
      decmsg = decmsg + (char) (encmsg.charAt(i) ^ key.charAt(j));
      j++;
      if(j==8) j = 0;
    }

    System.out.print("Decoded message: ");
    System.out.println(decmsg);
  }
}
```

8 . Can the bitwise operators be applied to the **double** type?

No.

9 . Show how this sequence can be rewritten using the **?** operator.

```
if(x < 0) y = 10;
else y = 20;
```

Here is the answer:

```
y = x < 0 ? 10 : 20;
```

10. In the following fragment, is the **&** a bitwise or logical operator? Why?

```
boolean a, b;
// ...
if(a & b) ...
```

It is a logical operator because the operands are of type **boolean**.

11. Is it an error to overrun the end of an array?

Yes.

Is it an error to index an array with a negative value?

Yes. All array indexes start at zero.

12. What is the unsigned right-shift operator?

```
>>>
```

13. Rewrite the **MinMax** class shown earlier in this chapter so that it uses a for-each style **for** loop.

```
// Find the minimum and maximum values in an array.
class MinMax {
  public static void main(String args[]) {
    int nums[] = new int[10];
    int min, max;

    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
    nums[9] = 49;

    min = max = nums[0];
    for(int v : nums) {
      if(v < min) min = v;
      if(v > max) max = v;
    }
    System.out.println("min and max: " + min + " " + max);
  }
}
```

14. Can the **for** loops that perform sorting in the **Bubble** class shown in Try This 5-1 be converted into for-each style loops? If not, why not?

No, the **for** loops in the **Bubble** class that perform the sort cannot be converted into for-each style loops. In the case of the outer loop, the current value of its loop counter is needed by the inner loop. In the case of the inner loop, out-of-order values must be exchanged, which implies assignments. Assignments to the underlying array cannot take place when using a for-each style loop.

15. Can a **String** control a **switch** statement?

1 . Given this fragment,

```
class X {
  private int count;
```

is the following fragment correct?

```
class Y {
  public static void main(String args[]) {
    X ob = new X();

    ob.count = 10;
```

No; a **private** member cannot be accessed outside of its class.

2 . An access modifier must _____ a member's declaration.

precede

3 . The complement of a queue is a stack. It uses first-in, last-out accessing and is often likened to a stack of plates. The first plate put on the table is the last plate used. Create a stack class called **Stack** that can hold characters. Call the methods that access the stack **push( )** and **pop( )**. Allow the user to specify the size of the stack when it is created. Keep all other members of the **Stack** class private. (Hint: You can use the **Queue** class as a model; just change the way that the data is accessed.)

```
// A stack class for characters.
class Stack {
  private char stck[]; // this array holds the stack
  private int tos; // top of stack

  // Construct an empty Stack given its size.
  Stack(int size) {
    stck = new char[size]; // allocate memory for stack
    tos = 0;
  }

  // Construct a Stack from a Stack.
  Stack(Stack ob) {
    tos = ob.tos;
    stck = new char[ob.stck.length];

    // copy elements
    for(int i=0; i < tos; i++)
      stck[i] = ob.stck[i];
  }

  // Construct a stack with initial values.
  Stack(char a[]) {
    stck = new char[a.length];

    for(int i = 0; i < a.length; i++) {
      push(a[i]);
    }
  }

  // Push characters onto the stack.
  void push(char ch) {
    if(tos==stck.length) {
      System.out.println(" -- Stack is full.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Pop a character from the stack.
  char pop() {
    if(tos==0) {
```

```
      System.out.println(" -- Stack is empty.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }
}

// Demonstrate the Stack class.
class SDemo {
  public static void main(String args[]) {
    // construct 10-element empty stack
    Stack stk1 = new Stack(10);

    char name[] = {'T', 'o', 'm'};

    // construct stack from array
    Stack stk2 = new Stack(name);

    char ch;
    int i;

    // put some characters into stk1
    for(i=0; i < 10; i++)
      stk1.push((char) ('A' + i));

    // construct stack from another stack
    Stack stk3 = new Stack(stk1);

    // show the stacks.
    System.out.print("Contents of stk1: ");
    for(i=0; i < 10; i++) {
      ch = stk1.pop();
      System.out.print(ch);
    }

    System.out.println("\n");

    System.out.print("Contents of stk2: ");
    for(i=0; i < 3; i++) {
      ch = stk2.pop();
      System.out.print(ch);
    }

    System.out.println("\n");
    System.out.print("Contents of stk3: ");
      for(i=0; i < 10; i++) {
        ch = stk3.pop();
        System.out.print(ch);
      }
  }
}
```

Here is the output from the program:

```
Contents of stk1: JIHGFEDCBA
Contents of stk2: moT
Contents of stk3: JIHGFEDCBA
```

4 . Given this class,

```
class Test {
  int a;
  Test(int i) { a = i; }
}
```

write a method called **swap( )** that exchanges the contents of the objects referred to by two **Test** object references.

```
void swap(Test ob1, Test ob2) {
  int t;

  t = ob1.a;
  ob1.a = ob2.a;
  ob2.a = t;
}
```

5 . Is the following fragment correct?

```
class X {
  int meth(int a, int b) { ... }
  String meth(int a, int b) { ... }
```

6 . Write a recursive method that displays the contents of a string backwards.

```
// Display a string backwards using recursion.
class Backwards {
  String str;

  Backwards(String s) {
    str = s;
  }

  void backward(int idx) {
    if(idx != str.length()-1) backward(idx+1);

    System.out.print(str.charAt(idx));
  }
}

class BWDemo {
  public static void main(String args[]) {
    Backwards s = new Backwards("This is a test");

    s.backward(0);
  }
}
```

7 . If all objects of a class need to share the same variable, how must you declare that variable?

Shared variables are declared as **static**.

8 . Why might you need to use a **static** block?

A **static** block is used to perform any initializations related to the class, before any objects are created.

9 . What is an inner class?

An inner class is a nonstatic nested class.

10. To make a member accessible by only other members of its class, what access modifier must be used?

private

11. The name of a method plus its parameter list constitutes the method's _____.

signature

12. An **int** argument is passed to a method by using call-by-_____.

value

13. Create a varargs method called **sum( )** that sums the **int** values passed to it. Have it return the result. Demonstrate its use.

There are many ways to craft the solution. Here is one:

```
class SumIt {
  int sum(int ... n) {
    int result = 0;

    for(int i = 0; i < n.length; i++)
      result += n[i];
    return result;
  }
}

class SumDemo {
  public static void main(String args[]) {

    SumIt siObj = new SumIt();

    int total = siObj.sum(1, 2, 3);
    System.out.println("Sum is " + total);

    total = siObj.sum(1, 2, 3, 4, 5);
    System.out.println("Sum is " + total);
  }
}
```

14. Can a varargs method be overloaded?

Yes.

15. Show an example of an overloaded varargs method that is ambiguous.

Here is one example of an overloaded varargs method that is ambiguous:

```
double myMeth(double ... v ) { // ...

double myMeth(double d, double ... v) { // ...
```

If you try to call **myMeth( )** with one argument, like this,

```
myMeth(1.1);
```

1 . Does a superclass have access to the members of a subclass?

Does a subclass have access to the members of a superclass?

No, a superclass has no knowledge of its subclasses. Yes, a subclass has access to all nonprivate members of its superclass.

2 . Create a subclass of **TwoDShape** called **Circle**. Include an **area( )** method that computes the area of the circle and a constructor that uses **super** to initialize the **TwoDShape** portion.

```
// A subclass of TwoDShape for circles.
class Circle extends TwoDShape {
  // A default constructor.
  Circle() {
    super();
}
  // Construct Circle
  Circle(double x) {
    super(x, "circle"); // call superclass constructor
  }

  // Construct an object from an object.
  Circle(Circle ob) {
    super(ob); // pass object to TwoDShape constructor
  }

  double area() {
    return (getWidth() / 2) * (getWidth() / 2) * 3.1416;
  }
}
```

3 . How do you prevent a subclass from having access to a member of a superclass?

To prevent a subclass from having access to a superclass member, declare that member as **private**.

4 . Describe the purpose and use of the two versions of **super** described in this chapter.

The **super** keyword has two forms. The first is used to call a superclass constructor. The general form of this usage is

super (*param-list*);

The second form of **super** is used to access a superclass member. It has this general form:

super.*member*

5 . Given the following hierarchy, in what order do the constructors for these classes complete their execution when a **Gamma** object is instantiated?

```
class Alpha { ...

class Beta extends Alpha { ...

Class Gamma extends Beta { ...
```

Constructors complete their execution in order of derivation. Thus, when a **Gamma** object is created, the order is **Alpha**, **Beta**, **Gamma**.

6 . A superclass reference can refer to a subclass object. Explain why this is important as it is related to method overriding.

When an overridden method is called through a superclass reference, it is the type of the object being referred to that determines which version of the method is called.

7 . What is an abstract class?

An abstract class contains at least one abstract method.

8 . How do you prevent a method from being overridden? How do you prevent a class from being inherited?

To prevent a method from being overridden, declare it as **final**. To prevent a class from being inherited, declare it as **final**.

9 . Explain how inheritance, method overriding, and abstract classes are used to support polymorphism.

Inheritance, method overriding, and abstract classes support polymorphism by enabling you to create a generalized class structure that can be implemented by a variety of classes. Thus, the abstract class defines a consistent interface that is shared by all implementing classes. This embodies the concept of "one interface, multiple methods."

10. What class is a superclass of every other class?

The **Object** class.

11. A class that contains at least one abstract method must, itself,

12. What keyword is used to create a named constant?

final

Chapter 8: Packages and Interfaces

1. Using the code from Try This 8-1, put the **ICharQ** interface and its three implementations into a package called **qpack**. Keeping the queue demonstration class **IQDemo** in the default package, show how to import and use the classes in **qpack**.

To put **ICharQ** and its implementations into the **qpack** package, you must separate each into its own file, make each implementation class **public**, and add this statement to the top of each file.

```
package qpack;
```

Once this has been done, you can use **qpack** by adding this **import** statement to **IQDemo**.

```
import qpack.*;
```

2. What is a namespace? Why is it important that Java allows you to partition the namespace?

A namespace is a declarative region. By partitioning the namespace, you can prevent name collisions.

3. Packages are stored in _____.

directories

4. Explain the difference between **protected** and default access.

A member with **protected** access can be used within its package and by a subclass in any package.

A member with default access can be used only within its package.

5. Explain the two ways that the members of a package can be used by other packages.

To use a member of a package, you can either fully qualify its name, or you can import it using **import**.

6. "One interface, multiple methods" is a key tenet of Java. What feature best exemplifies it?

The interface best exemplifies the one interface, multiple methods principle of OOP.

7. How many classes can implement an interface? How many interfaces can a class implement?

An interface can be implemented by an unlimited number of classes. A class can implement as many interfaces as it chooses.

8. Can interfaces be extended?

Yes, interfaces can be extended.

9. Create an interface for the **Vehicle** class from Chapter 7. Call the interface **IVehicle**.

```
interface IVehicle {

  // Return the range.
  int range();

  // Compute fuel needed for a given distance.
  double fuelneeded(int miles);

  // Access methods for instance variables.
  int getPassengers();
  void setPassengers(int p);
  int getFuelcap();
  void setFuelcap(int f);
  int getMpg();
  void setMpg(int m);
}
```

10. Variables declared in an interface are implicitly **static** and **final**. Can they be shared with other parts of a program?

Yes, interface variables can be used as named constants that are shared by all files in a program. They are brought into view by importing their interface.

11. A package is, in essence, a container for classes. True or False?

True.

12. What standard Java package is automatically imported into a program?

default

14. Beginning with JDK 8, is it possible to define a **static** method in an **interface**?

Yes

15. Assume that the **ICharQ** interface shown in Try This 8-1 has been in widespread use for several years. Now, you want to add a method to it called **reset( )**, which will be used to reset the queue to its empty, starting condition. Assuming JDK 8 or later, how can this be accomplished without breaking preexisting code?

To avoid breaking preexisting code, you must use a default interface method. Because you can't know how to reset each queue implementation, the default **reset( )** implementation will need to report an error that indicates that it is not implemented. (The best way to do this is to use an exception. Exceptions are examined in the following chapter.) Fortunately, since no preexisting code assumes that **ICharQ** defines a **reset( )** method, no preexisting code will encounter that error, and no preexisting code will be broken.

16. How is a **static** method in an interface called?

A **static** interface method is called through its interface name, by use of the dot operator.

Chapter 9: Exception Handling

1 . What class is at the top of the exception hierarchy?

Throwable is at the top of the exception hierarchy.

2 . Briefly explain how to use **try** and **catch**.

The **try** and **catch** statements work together. Program statements that you want to monitor for exceptions are contained within a **try** block. An exception is caught using **catch**.

3 . What is wrong with this fragment?

```
// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
  // handle error
}
```

There is no **try** block preceding the **catch** statement.

4 . What happens if an exception is not caught?

If an exception is not caught, abnormal program termination results.

5 . What is wrong with this fragment?

```
class A extends Exception { ...

class B extends A { ...

// ...
try {
  // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

In the fragment, a superclass **catch** precedes a subclass **catch**. Since the superclass **catch** will catch all subclasses too, unreachable code is created.

6 . Can an inner **catch** rethrow an exception to an outer **catch**?

Yes, an exception can be rethrown.

7 . The **finally** block is the last bit of code executed before your program ends. True or False? Explain your answer.

False. The **finally** block is the code executed when a **try** block ends.

8 . What type of exceptions must be explicitly declared in a **throws** clause of a method?

All exceptions except those of type **RuntimeException** and **Error** must be declared in a **throws** clause.

9 . What is wrong with this fragment?

```
class MyClass { // ... }
// ...
throw new MyClass();
```

MyClass does not extend **Throwable**. Only subclasses of **Throwable** can be thrown by **throw**.

10. In question 3 of the Chapter 6 Self Test, you created a **Stack** class. Add custom exceptions to your class that report stack full and stack empty conditions.

```
// An exception for stack-full errors.
class StackFullException extends Exception {
  int size;

  StackFullException(int s) { size = s; }

  public String toString() {
   return "\nStack is full. Maximum size is " +
          size;
  }
}

// An exception for stack-empty errors.
class StackEmptyException extends Exception {

  public String toString() {
    return "\nStack is empty.";
  }
}
// A stack class for characters.
class Stack {
  private char stck[]; // this array holds the stack
  private int tos; // top of stack

  // Construct an empty Stack given its size.
  Stack(int size) {
    stck = new char[size]; // allocate memory for stack
    tos = 0;
  }

  // Construct a Stack from a Stack.
  Stack(Stack ob) {
    tos = ob.tos;
    stck = new char[ob.stck.length];

    // copy elements
    for(int i=0; i < tos; i++)
      stck[i] = ob.stck[i];
  }

  // Construct a stack with initial values.
  Stack(char a[]) {
    stck = new char[a.length];

    for(int i = 0; i < a.length; i++) {
      try {
        push(a[i]);
      }
      catch(StackFullException exc) {
        System.out.println(exc);
      }
    }
  }

  // Push characters onto the stack.
  void push(char ch) throws StackFullException {
    if(tos==stck.length)
      throw new StackFullException(stck.length);

    stck[tos] = ch;
    tos++;
  }

  // Pop a character from the stack.
  char pop() throws StackEmptyException {
    if(tos==0)
      throw new StackEmptyException();
    tos--;
    return stck[tos];
  }
}
```

11. What are the three ways that an exception can be generated?

An exception can be generated by an error in the JVM, by an error in your program, or explicitly via a **throw** statement.

12. What are the two direct subclasses of **Throwable**?

Error and **Exception**

13. What is the multi-catch feature?

The multi-catch feature allows one **catch** clause to catch two or more exceptions.

14. Should your code typically catch exceptions of type **Error**?

No.

Chapter 10: Using I/O

The byte streams are the original streams defined by Java. They are especially useful for binary I/O, and they support random-access files. The character streams are optimized for Unicode.

2 . Even though console input and output is text-based, why does Java still use byte streams for this purpose?

The predefined streams, **System.in**, **System.out**, and **System.err**, were defined before Java added the character streams.

3 . Show how to open a file for reading bytes.

Here is one way to open a file for **byte** input:

```
FileInputStream fin = new FileInputStream("test");
```

4 . Show how to open a file for reading characters.

Here is one way to open a file for reading characters:

```
FileReader fr = new FileReader("test");
```

5 . Show how to open a file for random-access I/O.

Here is one way to open a file for random access:

```
randfile = new RandomAccessFile("test", "rw");
```

6 . How do you convert a numeric string such as "123.23" into its binary equivalent?

To convert numeric strings into their binary equivalents, use the parsing methods defined by the type wrappers, such as **Integer** or **Double**.

7 . Write a program that copies a text file. In the process, have it convert all spaces into hyphens. Use the byte stream file classes. Use the traditional approach to closing a file by explicitly calling **close( )**.

```
/* Copy a text file, substituting hyphens for spaces.

   This version uses byte streams.

   To use this program, specify the name
   of the source file and the destination file.
   For example,

   java Hyphen source target
*/

import java.io.*;

class Hyphen {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin = null;
    FileOutputStream fout = null;

    // First make sure that both files have been specified.
    if(args.length !=2 ) {
      System.out.println("Usage: Hyphen From To");
      return;
    }

    // Copy file and substitute hyphens.
    try {
      fin = new FileInputStream(args[0]);
      fout = new FileOutputStream(args[1]);

      do {
        i = fin.read();

        // convert space to a hyphen
        if((char)i == ' ') i = '-';

        if(i != -1) fout.write(i);
      } while(i != -1);
    } catch(IOException exc) {
      System.out.println("I/O Error: " + exc);
    } finally {
      try {
        if(fin != null) fin.close();
      } catch(IOException exc) {
        System.out.println("Error closing input file.");
      }

      try {
        if(fin != null) fout.close();
      } catch(IOException exc) {
        System.out.println("Error closing output file.");
      }
    }
  }
}
```

8 . Rewrite the program in question 7 so that it uses the character stream classes. This time, use the **try**-with-resources statement to automatically close the file.

```
/* Copy a text file, substituting hyphens for spaces.

   This version uses character streams.

   To use this program, specify the name
   of the source file and the destination file.
   For example,

   java Hyphen2 source target

*/

import java.io.*;

class Hyphen2 {
  public static void main(String args[])
    throws IOException
  {
    int i;

    // First make sure that both files have been specified.
    if(args.length !=2 ) {
      System.out.println("Usage: CopyFile From To");
      return;
    }

    // Copy file and substitute hyphens.
    // Use the try-with-resources statement.
    try (FileReader fin = new FileReader(args[0]);
         FileWriter fout = new FileWriter(args[1]))
    {
      do {
        i = fin.read();

        // convert space to a hyphen
        if((char)i == ' ') i = '-';

        if(i != -1) fout.write(i);
      } while(i != -1);
    } catch(IOException exc) {
      System.out.println("I/O Error: " + exc);
    }
  }
}
```

9 . What type of stream is **System.in**?

InputStream

10. What does the **read( )** method of **InputStream** return when the end of the stream is reached?

–1

11. What type of stream is used to read binary data?

DataInputStream

12. **Reader** and **Writer** are at the top of the _____ class hierarchies.

character-based I/O

13. The **try**-with-resources statement is used for _____ _____.

automatic resource management

14. If you are using the traditional method of closing a file, then closing a file within a **finally** block is generally a good approach. True or False?

True

Chapter 11: Multithreaded Programming

1 . How does Java's multithreading capability enable you to write more efficient programs?

Multithreading allows you to take advantage of the idle time that is present in nearly all programs. When one thread can't run, another can. In multicore systems, two or more threads can execute simultaneously.

2 . Multithreading is supported by the _____ class and the _____ interface.

Multithreading is supported by the **Thread** class and the **Runnable** interface.

3 . When creating a runnable object, why might you want to extend **Thread** rather than implement **Runnable**?

You will extend **Thread** when you want to override one or more of **Thread**'s methods other than **run( )**.

4 . Show how to use **join( )** to wait for a thread object called **MyThrd** to end.

```
MyThrd.join();
```

5 . Show how to set a thread called **MyThrd** to three levels above normal priority.

```
MyThrd.setPriority(Thread.NORM_PRIORITY+3);
```

6 . What is the effect of adding the **synchronized** keyword to

Adding **synchronized** to a method allows only one thread at a time to use the method for any given object of its class.

7 . The **wait( )** and **notify( )** methods are used to perform _____.

interthread communication

8 . Change the **TickTock** class so that it actually keeps time.

That is, have each tick take one half second, and each tock take one half second. Thus, each tick-tock will take one second. (Don't worry about the time it takes to switch tasks, etc.)

To make the **TickTock** class actually keep time, simply add calls to **sleep( )**, as shown here:

```
// Make the TickTock class actually keep time.

class TickTock {

  String state; // contains the state of the clock

  synchronized void tick(boolean running) {
    if(!running) { // stop the clock
      state = "ticked";
      notify(); // notify any waiting threads
      return;
    }

    System.out.print("Tick ");

    // wait 1/2 second
    try {
      Thread.sleep(500);
    } catch(InterruptedException exc) {
      System.out.println("Thread interrupted.");
    }

    state = "ticked"; // set the current state to ticked

    notify(); // let tock() run
    try {
      while(!state.equals("tocked"))
        wait(); // wait for tock() to complete
    }
    catch(InterruptedException exc) {
      System.out.println("Thread interrupted.");
    }
  }

  synchronized void tock(boolean running) {
    if(!running) { // stop the clock
      state = "tocked";
      notify(); // notify any waiting threads
      return;
    }

    System.out.println("Tock");

    // wait 1/2 second
    try {
      Thread.sleep(500);
    } catch(InterruptedException exc) {
      System.out.println("Thread interrupted.");
    }

    state = "tocked"; // set the current state to tocked

    notify(); // let tick() run
    try {
      while(!state.equals("ticked"))
        wait(); // wait for tick to complete
    }
    catch(InterruptedException exc) {
      System.out.println("Thread interrupted.");
    }
  }
}
```

9 . Why can't you use **suspend( )**, **resume( )**, and **stop( )** for new programs?

The **suspend( )**, **resume( )**, and **stop( )** methods have been deprecated because they can cause serious run-time problems.

10. What method defined by **Thread** obtains the name of a thread?

getName( )

11. What does **isAlive( )** return?

It returns **true** if the invoking thread is still running, and **false** if it has been terminated.

Chapter 12: Enumerations, Autoboxing, Static Import, and Annotations

1 . Enumeration constants are said to be *self-typed*. What does this mean?

In the term *self-typed*, the "self" refers to the type of the enumeration in which the constant is defined. Thus, an enumeration constant is an object of the enumeration of which it is a part.

3. Given the following enumeration, write a program that uses **values( )** to show a list of the constants and their ordinal values.

```
enum Tools {
  SCREWDRIVER, WRENCH, HAMMER, PLIERS
}
```

The solution is

```
enum Tools {
  SCREWDRIVER, WRENCH, HAMMER, PLIERS
}

class ShowEnum {
  public static void main(String args[]) {
    for(Tools d : Tools.values())
      System.out.print(d + " has ordinal value of " +
                       d.ordinal() + '\n');
  }
}
```

4. The traffic light simulation developed in Try This 12-1 can be improved with a few simple changes that take advantage of an enumeration's class features. In the version shown, the duration of each color was controlled by the **TrafficLightSimulator** class by hard-coding these values into the **run( )** method. Change this so that the duration of each color is stored by the constants in the **TrafficLightColor** enumeration. To do this, you will need to add a constructor, a private instance variable, and a method called **getDelay( )**. After making these changes, what improvements do you see? On your own, can you think of other improvements? (Hint: Try using ordinal values to switch light colors rather than relying on a **switch** statement.)

The improved version of the traffic light simulation is shown here. There are two major improvements. First, a light's delay is now linked with its enumeration value, which gives more structure to the code. Second, the **run( )** method no longer needs to use a **switch** statement to determine the length of the delay. Instead, **sleep( )** is passed **tlc.getDelay( )**, which causes the delay associated with the current color to be used automatically.

```
// An improved version of the traffic light simulation that
// stores the light delay in TrafficLightColor.

// An enumeration of the colors of a traffic light.
enum TrafficLightColor {
  RED(12000), GREEN(10000), YELLOW(2000);

  private int delay;

  TrafficLightColor(int d) {
    delay = d;
  }

  int getDelay() { return delay; }
}

// A computerized traffic light.
class TrafficLightSimulator implements Runnable {
  private Thread thrd; // holds the thread that runs the simulation
  private TrafficLightColor tlc; // holds the current traffic light color
  boolean stop = false; // set to true to stop the simulation
  boolean changed = false; // true when the light has changed

  TrafficLightSimulator(TrafficLightColor init) {
    tlc = init;

    thrd = new Thread(this);
    thrd.start();
  }

  TrafficLightSimulator() {
    tlc = TrafficLightColor.RED;

    thrd = new Thread(this);
    thrd.start();
  }

  // Start up the light.
  public void run() {
    while(!stop) {
      // Notice how this code has been simplified!
      try {
        Thread.sleep(tlc.getDelay());
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }

      changeColor();
    }
  }

  // Change color.
  synchronized void changeColor() {
    switch(tlc) {
      case RED:
        tlc = TrafficLightColor.GREEN;
        break;
      case YELLOW:
        tlc = TrafficLightColor.RED;
        break;
      case GREEN:
        tlc = TrafficLightColor.YELLOW;
    }

    changed = true;
    notify(); // signal that the light has changed
  }
```

```
  // Wait until a light change occurs.
  synchronized void waitForChange() {
    try {
      while(!changed)
        wait(); // wait for light to change
      changed = false;
    } catch(InterruptedException exc) {
      System.out.println(exc);
    }
  }

  // Return current color.
  synchronized TrafficLightColor getColor() {
    return tlc;
  }

  // Stop the traffic light.
  synchronized void cancel() {
    stop = true;
  }
}

class TrafficLightDemo {
  public static void main(String args[]) {
    TrafficLightSimulator tl =
      new TrafficLightSimulator(TrafficLightColor.GREEN);

    for(int i=0; i < 9; i++) {
      System.out.println(tl.getColor());
      tl.waitForChange();
    }

    tl.cancel();
  }
}
```

5 . Define boxing and unboxing. How does autoboxing/unboxing affect these actions?

Boxing is the process of storing a primitive value in a type wrapper object. Unboxing is the process of retrieving the primitive value from the type wrapper. Autoboxing automatically boxes a primitive value without having to explicitly construct an object. Auto-unboxing automatically retrieves the primitive value from a type wrapper without having to explicitly call a method, such as **intValue( )**.

6 . Change the following fragment so that it uses autoboxing.

```
Short val = new Short(123);
```

The solution is

```
Short val = 123;
```

7 . In your own words, what does static import do?

Static import brings into the global namespace the static members of a class or interface. This means that static members can be used without having to be qualified by their class or interface name.

8 . What does this statement do?

```
import static java.lang.Integer.parseInt;
```

The statement brings into the global namespace the **parseInt( )** method of the type wrapper **Integer**.

9 . Is static import designed for special-case situations, or is it good practice to bring all static members of all classes into view?

Static import is designed for special cases. Bringing many static members into view will lead to namespace collisions and destructure your code.

10. An annotation is syntactically based on a/an _____.

interface

11. What is a marker annotation?

A marker annotation is one that does not take arguments.

12. An annotation can be applied only to methods. True or False?

False. Any type of declaration can have an annotation. Beginning with JDK 8, a type use can also have an annotation.

Chapter 13: Generics

1 . Generics are important to Java because they enable the creation of code that is

A. Type-safe

B. Reusable

C. Reliable

D. All of the above

D. All of the above

2 . Can a primitive type be used as a type argument?

3 . Show how to declare a class called **FlightSched** that takes two generic parameters.

The solution is

4 . Beginning with your answer to question 3, change **FlightSched**'s second type parameter so that it must extend **Thread**.

The solution is

5 . Now, change **FlightSched** so that its second type parameter must be a subclass of its first type parameter.

The solution is

```
class FlightSched<T, V extends T> {
```

6 . As it relates to generics, what is the **?** and what does it do?

The **?** is the wildcard argument. It matches any valid type.

7 . Can the wildcard argument be bounded?

Yes, a wildcard can have either an upper or lower bound.

8 . A generic method called **MyGen( )** has one type parameter. Furthermore, **MyGen( )** has one parameter whose type is that of the type parameter. It also returns an object of that type parameter. Show how to declare **MyGen( )**.

The solution is

```
<T> T MyGen(T o) { // ...
```

9 . Given this generic interface

show the declaration of a class called **MyClass** that implements **IGenIF**.

The solution is

```
class MyClass<T, V extends T> implements IGenIF<T, V> { // ...
```

10. Given a generic class called **Counter<T>**, show how to create an object of its raw type.

To obtain **Counter<T>**'s raw type, simply use its name without any type specification, as shown here:

```
Counter x = new Counter;
```

11. Do type parameters exist at run time?

No. All type parameters are erased during compilation, and appropriate casts are substituted. This process is called erasure.

12. Convert your solution to question 10 of the Self Test for Chapter 9 so that it is generic. In the process, create a stack interface called **IGenStack** that generically defines the operations **push( )** and **pop( )**.

```
// A generic stack.

interface IGenStack<T> {
  void push(T obj) throws StackFullException;
  T pop() throws StackEmptyException;
}

// An exception for stack-full errors.
class StackFullException extends Exception {
  int size;

  StackFullException(int s) { size = s; }
```

```
  public String toString() {
   return "\nStack is full. Maximum size is " +
          size;
  }
}

// An exception for stack-empty errors.
class StackEmptyException extends Exception {

  public String toString() {
   return "\nStack is empty.";
  }
}

// A stack class for characters.
class GenStack<T> implements IGenStack<T> {
  private T stck[]; // this array holds the stack
  private int tos; // top of stack

  // Construct an empty stack given its size.
  GenStack(T[] stckArray) {
    stck = stckArray;
    tos = 0;
  }

  // Construct a stack from a stack.
  GenStack(T[] stckArray, GenStack<T> ob) {
    tos = ob.tos;
    stck = stckArray;

    try {
      if(stck.length < ob.stck.length)
        throw new StackFullException(stck.length);
    }
    catch(StackFullException exc) {
      System.out.println(exc);
    }

    // Copy elements.
    for(int i=0; i < tos; i++)
      stck[i] = ob.stck[i];
  }

  // Construct a stack with initial values.
  GenStack(T[] stckArray, T[] a) {
    stck = stckArray;

    for(int i = 0; i < a.length; i++) {
      try {
        push(a[i]);
      }
      catch(StackFullException exc) {
        System.out.println(exc);
      }
    }
  }

  // Push objects onto the stack.
  public void push(T obj) throws StackFullException {
    if(tos==stck.length)
      throw new StackFullException(stck.length);

    stck[tos] = obj;
    tos++;
  }

  // Pop an object from the stack.
  public T pop() throws StackEmptyException {
    if(tos==0)
      throw new StackEmptyException();

    tos--;
    return stck[tos];
  }
}

// Demonstrate the GenStack class.
class GenStackDemo {
  public static void main(String args[]) {
    // Construct 10-element empty Integer stack.
    Integer iStore[] = new Integer[10];
    GenStack<Integer> stk1 = new GenStack<Integer>(iStore);

    // Construct stack from array.
    String name[] = {"One", "Two", "Three"};
    String strStore[] = new String[3];
    GenStack<String> stk2 =
        new GenStack<String>(strStore, name);

    String str;
    int n;

    try {
      // Put some values into stk1.
      for(int i=0; i < 10; i++)
        stk1.push(i);
```

13. What is < >?

The diamond operator.

14. How can the following be simplified?

```
MyClass<Double,String> obj = new MyClass<Double,String>(1.1,"Hi");
```

It can be simplified by use of the diamond operator as shown here:

1 . What is the lambda operator?

The lambda operator is **->**.

2 . What is a functional interface?

A functional interface is an interface that contains one and only one abstract method.

3 . How do functional interfaces and lambda expressions relate?

A lambda expression provides the implementation for the abstract method defined by the functional interface. The functional interface defines the target type.

4 . What are the two general types of lambda expressions?

The two types of lambda expressions are expression lambdas and block lambdas. An expression lambda specifies a single expression, whose value is returned by the lambda. A block lambda contains a block of code. Its value is specified by a **return** statement.

5 . Show a lambda expression that returns **true** if a number is between 10 and 20, inclusive.

6 . Create a functional interface that can support the lambda expression you created in question 5. Call the interface **MyTest** and its abstract method **testing( )**.

7 . Create a block lambda that computes the factorial of an integer value. Demonstrate its use. Use **NumericFunc**, shown in this chapter, for the functional interface.

```
interface NumericFunc {
  int func(int n);
}

class FactorialLambdaDemo {
  public static void main(String args[])
  {

    // This block lambda computes the factorial of an int value.
    NumericFunc factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;
      return result;
    };

    System.out.println("The factorial of 3 is " + factorial.func(3));
    System.out.println("The factorial of 3 is " + factorial.func(5));
    System.out.println("The factorial of 5 is " + factorial.func(9));
  }
}
```

8 . Create a generic functional interface called **MyFunc<T>**. Call its abstract method **func( )**. Have **func( )** return a reference of type **T**. Have it take a parameter of type **T**. (Thus, **MyFunc** will be a generic version of **NumericFunc** shown in the chapter.) Demonstrate its use by rewriting your answer to 7 so it uses **MyFunc<T>** rather than **NumericFunc**.

9 . Using the program shown in Try This 14-1, create a lambda expression that removes all spaces from a string and returns the result. Demonstrate this method by passing it to **changeStr( )**.

Here is the lambda expression that removes spaces. It is used to initialize the **remove** reference variable.

Here is an example of its use:

10. Can a lambda expression use a local variable? If so, what constraint must be met?

Yes, but the variable must be effectively **final**.

11. If a lambda expression throws a checked exception, the abstract method in the functional interface must have a **throws** clause that includes that exception. True or False?

True

12. What is a method reference?

A method reference is a way to refer to a method without executing it.

13. *When evaluated, a method reference creates an instance of the* supplied by its target context.

14. Given a class called **MyClass** that contains a **static** method called **myStaticMethod( )**, show how to specify a method reference to **myStaticMethod( )**.

15. Given a class called **MyClass** that contains an instance method called **myInstMethod( )** and assuming an object of **MyClass** called **mcObj**, show how to create a method reference to **myInstMethod( )** on **mcObj**.

16. To the **MethodRefDemo2** program, add a new method to **MyIntNum** called **hasCommonFactor( )**. Have it return **true** if its **int** argument and the value stored in the invoking **MyIntNum** object have at least one factor in common. For example, 9 and 12 have a common factor, which is 3, but 9 and 16 have no common factor. Demonstrate **hasCommonFactor( )** via a method reference.

Here is **MyIntNum** with the **hasCommonFactor( )** method added:

Here is an example of its use through a method reference:

17. How is a constructor reference specified?

A constructor reference is created by specifying the class name followed by **::** followed by **new**. For example, **MyClass::new**.

18. Java defines several predefined functional interfaces in what package?

java.util.function

Chapter 15: Applets, Events, and Miscellaneous Topics

1 . What method is called when an applet first begins running?

What method is called when an applet is removed from the system?

When an applet begins, the first method called is **init( )**.When an applet is removed, **destroy( )** is called.

2 . Explain why an applet must use multithreading if it needs to run continually.

An applet must use multithreading if it needs to run continually because applets are event-driven programs which must not enter a "mode" of operation. For example, if **start( )** never returns, then **paint( )** will never be called.

3 . Enhance Try This 15-1 so that it displays the string passed to it as a parameter. Add a second parameter that specifies the time delay (in milliseconds) between each rotation.

```
/* A simple banner applet that uses parameters.

*/
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
```

```
<param name=message value=" I like Java! ">
<param name=delay value=500>
</applet>
*/

public class ParamBanner extends Applet implements Runnable {
  String msg;
  int delay;
  Thread t;
  boolean stopFlag;

  // Initialize t to null.
  public void init() {
    String temp;

    msg = getParameter("message");
    if(msg == null) msg = " Java Rules the Web ";

    temp = getParameter("delay");

    try {
      if(temp != null)
        delay = Integer.parseInt(temp);
      else
        delay = 250; // default if not specified
    } catch(NumberFormatException exc) {
        delay = 250 ; // default on error
    }

    t = null;
  }

  // Start thread
  public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {
    // Redisplay banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(delay);
        if(stopFlag)
          break;
      } catch(InterruptedException exc) {}
    }
  }

  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;
    g.drawString(msg, 50, 30);
  }
}
```

4 . Extra challenge: Create an applet that displays the current time, updated once per second.

To accomplish this, you will need to do a little research. Here is a hint to help you get started: One way to obtain the current time is to use a **Calendar** object, which is part of the **java.util** package. (Remember, Oracle provides online documentation for all of Java's standard classes.) You should now be at the point where you can examine the **Calendar** class on your own and use its methods to solve this problem.

```
// A simple clock applet.

import java.util.*;
import java.awt.*;
import java.applet.*;
/*
<object code="Clock" width=200 height=50>
</object>
*/

public class Clock extends Applet implements Runnable {
  String msg;
  Thread t;
  Calendar clock;

  boolean stopFlag;

  // Initialize
  public void init() {
```

```
    t = null;
    msg = "";
  }

  // Start thread
  public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the clock.
  public void run() {
    // Redisplay clock
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(1000);
        if(stopFlag)
          break;

      } catch(InterruptedException exc) {}
    }
  }

  // Pause the clock.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the clock.
  public void paint(Graphics g) {
    clock = Calendar.getInstance();

    msg = "Current time is " +
          Integer.toString(clock.get(Calendar.HOUR));
    msg = msg + ":" +
          Integer.toString(clock.get(Calendar.MINUTE));
    msg = msg + ":" +
          Integer.toString(clock.get(Calendar.SECOND));
    g.drawString(msg, 30, 30);
  }
}
```

5 . Briefly explain Java's delegation event model.

In the delegation event model, a *source* generates an event and sends it to one or more *listeners*. A listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

6 . Must an event listener register itself with a source?

Yes; a listener must register with a source to receive events.

7 . Extra challenge: Another of Java's display methods is **drawLine( )**. It draws a line in the currently selected color between two points. It is part of the **Graphics** class. Using **drawLine( )**, write a program that tracks mouse movement. If the button is pressed, have the program draw a continuous line until the mouse button is released.

```
/* Track mouse motion by drawing a line
   when a mouse button is pressed. */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TrackM" width=300 height=100>
</applet>
*/

public class TrackM extends Applet
  implements MouseListener, MouseMotionListener {

  int curX = 0, curY = 0; // current coordinates
  int oldX = 0, oldY = 0; // previous coordinates
  boolean draw;

  public void init() {
    addMouseListener(this);
    addMouseMotionListener(this);
    draw = false;
  }

  /* The next three methods are not used, but must
     be null-implemented because they are defined
     by MouseListener. */

  // Handle mouse entered.
  public void mouseEntered(MouseEvent me) {
  }

  // Handle mouse exited.
  public void mouseExited(MouseEvent me) {
  }

  // Handle mouse click.
  public void mouseClicked(MouseEvent me) {
  }
```

```
  // Handle button pressed.
  public void mousePressed(MouseEvent me) {
    // save coordinates
    oldX = me.getX();
    oldY = me.getY();
    draw = true;
  }

  // Handle button released.
  public void mouseReleased(MouseEvent me) {
    draw = false;
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    // save coordinates
    curX = me.getX();
    curY = me.getY();
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
  }

  // Display line in applet window.
  public void paint(Graphics g) {
    if(draw)
      g.drawLine(oldX, oldY, curX, curY);
  }
}
```

8 . Briefly describe the **assert** keyword.

The **assert** keyword creates an assertion, which is a condition that should be true during program execution. If the assertion is false, an **AssertionError** is thrown.

9 . Give one reason why a native method might be useful to some types of programs.

A native method is useful when interfacing to routines written in languages other than Java, or when optimizing code for a specific run-time environment.

Chapter 16: Introducing Swing

1 . In general, AWT components are heavyweight and Swing components are *lightweight*.

2 . Can the look and feel of a Swing component be changed? If so, what feature enables this?

Yes. Swing's pluggable look and feel is the feature that enables this.

3 . What is the most commonly used top-level container for an application?

JFrame

4 . Top-level containers have several panes. To what pane are components added?

Content pane

5 . Show how to construct a label that contains the message "Select an entry from the list".

```
JLabel("Select an entry from the list")
```

6 . All interaction with GUI components must take place on what thread?

event-dispatching thread

7 . What is the default action command associated with a **JButton**? How can the action command be changed?

The default action command string is the text shown inside the button. It can be changed by calling **setActionCommand( )**.

8 . What event is generated when a push button is pressed?

ActionEvent

9 . Show how to create a text field that has 32 columns.

```
JTextField(32)
```

10. Can a **JTextField** have its action command set? If so, how?

Yes, by calling **setActionCommand( )**.

11. What Swing component creates a check box? What event is generated when a check box is selected or deselected?

JCheckBox creates a check box. An **ItemEvent** is generated when a check box is selected or deselected.

12. ***JList*** *displays a list of items from* which the user can select. True or False?

13. What event is generated when the user selects or deselects an item in a **JList**?

ListSelectionEvent

14. What method sets the selection mode of a **JList**? What method obtains the index of the first selected item?

setSelectionMode( ) sets the selection mode. **getSelectedIndex( )** obtains the index of the first selected item.

15. To create a Swing-based applet, what class must you inherit?

JApplet

16. Usually, Swing-based applets use **invokeAndWait( )** to create the initial GUI. True or False?

True

17. Add a check box to the file comparer developed in Try This 16-1 that has the following text: Show position of mismatch. When this box is checked, have the program display the location of the first point in the files at which a mismatch occurs.

```
/*
    Try This 16-1

    A Swing-based file comparison utility.

    This version has a check box that causes the
    location of the first mismatch to be shown.

*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

class SwingFC implements ActionListener {

  JTextField jtfFirst;  // holds the first file name
  JTextField jtfSecond; // holds the second file name

  JButton jbtnComp; // button to compare the files

  JLabel jlabFirst, jlabSecond; // displays prompts
  JLabel jlabResult; // displays results and error messages

  JCheckBox jcbLoc; // check to display location of mismatch

  SwingFC() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("Compare Files");

    // Specify FlowLayout for the layout manager.
    jfrm.setLayout(new FlowLayout());

    // Give the frame an initial size.
    jfrm.setSize(200, 220);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create the text fields for the file names..
    jtfFirst = new JTextField(14);
    jtfSecond = new JTextField(14);
  // Set the action commands for the text fields.
  jtfFirst.setActionCommand("fileA");
  jtfSecond.setActionCommand("fileB");

  // Create the Compare button.
  JButton jbtnComp = new JButton("Compare");

  // Add action listener for the Compare button.
  jbtnComp.addActionListener(this);

  // Create the labels.
  jlabFirst = new JLabel("First file: ");
  jlabSecond = new JLabel("Second file: ");
  jlabResult = new JLabel("");

  // Create check box.
  jcbLoc = new JCheckBox("Show position of mismatch");

  // Add the components to the content pane.
  jfrm.add(jlabFirst);
  jfrm.add(jtfFirst);
  jfrm.add(jlabSecond);
  jfrm.add(jtfSecond);
  jfrm.add(jcbLoc);
  jfrm.add(jbtnComp);
  jfrm.add(jlabResult);

  // Display the frame.
  jfrm.setVisible(true);
}

// Compare the files when the Compare button is pressed.
public void actionPerformed(ActionEvent ae) {
  int i=0, j=0;
  int count = 0;

  // First, confirm that both file names have
  // been entered.
  if(jtfFirst.getText().equals("")) {
    jlabResult.setText("First file name missing.");
    return;
  }
  if(jtfSecond.getText().equals("")) {
    jlabResult.setText("Second file name missing.");
```

```
    // Compare files. Use try-with-resources to manage the files.
    try (FileInputStream f1 = new FileInputStream(jtfFirst.getText());
         FileInputStream f2 = new FileInputStream(jtfSecond.getText()))
    {
      // Check the contents of each file.
      do {
        i = f1.read();
        j = f2.read();
        if(i != j) break;
        count++;
      } while(i != -1 && j != -1);

      if(i != j) {
        if(jcbLoc.isSelected())
          jlabResult.setText("Files differ at location " + count);
        else
          jlabResult.setText("Files are not the same.");
      }
      else
        jlabResult.setText("Files compare equal.");

    } catch(IOException exc) {
      jlabResult.setText("File Error");
    }
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingFC();
      }
    });
  }
}
```

18. Change the **ListDemo** program so that it allows multiple items in the list to be selected.

```
// Demonstrate multiple selection in a JList.

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

class ListDemo implements ListSelectionListener {
  JList<String> jlst;
  JLabel jlab;
  JScrollPane jscrlp;

  // Create an array of names.
  String names[] = { "Sherry", "Jon", "Rachel",
                     "Sasha", "Josselyn", "Randy",
                     "Tom", "Mary", "Ken",
                     "Andrew", "Matt", "Todd" };

  ListDemo() {
    // Create a new JFrame container.
    JFrame jfrm = new JFrame("JList Demo");

    // Specify a flow Layout.
    jfrm.setLayout(new FlowLayout());

    // Give the frame an initial size.
    jfrm.setSize(200, 160);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a JList.
    jlst = new JList<String>(names);

    // By removing the following line, multiple selection (which
    // is the default behavior of a JList) will be used.
//    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add list to a scroll pane.
    jscrlp = new JScrollPane(jlst);

    // Set the preferred size of the scroll pane.
    jscrlp.setPreferredSize(new Dimension(120, 90));

    // Make a label that displays the selection.
    jlab = new JLabel("Please choose a name");

    // Add list selection handler.
    jlst.addListSelectionListener(this);

    // Add the list and label to the content pane.
    jfrm.add(jscrlp);
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle list selection events.
  public void valueChanged(ListSelectionEvent le) {
    // Get the indices of the changed item.
    int indices[] = jlst.getSelectedIndices();

    // Display the selections, if one or more items
    // were selected.
    if(indices.length != 0) {
      String who = "";

      // Construct a string of the names.
      for(int i : indices)
        who += names[i] + " ";

      jlab.setText("Current selections: " + who);
    }
    else // Otherwise, reprompt.
      jlab.setText("Please choose a name");
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new ListDemo();
      }
```

Chapter 17: Introducing JavaFX

1 . What is the top-level package name of the JavaFX framework?

javafx

2 . Two concepts central to JavaFX are a stage and a scene. What classes encapsulate them?

Stage and **Scene**

3 . A scene graph is composed of _____.

nodes

4 . The base class for all nodes is _____.

Node

5 . What class will all JavaFX applications extend?

Application

6 . What are the three JavaFX life-cycle methods?

init( ), **start( )**, and **stop( )**

7 . In what life-cycle method can you construct an application's stage?

start( )

8 . The **launch( )** method is called to start a free-standing JavaFX application. True or False?

True

9 . What are the names of the JavaFX classes that support a label and a button?

Label and **Button**

10. One way to terminate a free-standing JavaFX application is to call **Platform.exit( )**. **Platform** is packaged in **javafx.Application**. When called, **exit( )** immediately terminates the program. With this in mind, change the **JavaFXEventDemo** program shown in this chapter so that it has two buttons called Run and Exit. If Run is pressed, have the program display that choice in a label. If Exit is pressed, have the application terminate. Use lambda expressions for the event handlers.

```
// Demonstrate Platform.exit().

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {
```

```
    // Give the stage a title.
    myStage.setTitle("Use Platform.exit().");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 100);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    response = new Label("Push a Button");

    // Create two push buttons.
    Button btnRun = new Button("Run");
    Button btnExit = new Button("Exit");

    // Handle the action events for the Run button.
    btnRun.setOnAction((ae) -> response.setText("You pressed Run."));

    // Handle the action events for the Exit button.
    btnExit.setOnAction((ae) -> Platform.exit());

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(btnRun, btnExit, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

11. What **JavaFX** control implements a check box?

CheckBox

12. **ListView** is a control that displays a directory list of files on the local file system. True or False?

False. **ListView** displays of list of items from which the user can choose.

13. Convert the Swing-based file comparison program in Try This 16-1 so it uses JavaFX instead. In the process, make use of another of JavaFX's features: its ability to fire an action event on a button under program control. This is done by calling **fire( )** on the button instance. For example, assuming a **Button** called **myButttton**, the following will fire an action event on it: **myButton.fire( )**. Use this fact when implementing the event handlers for the text fields that hold the names of the files to compare. If the user presses ENTER when in either of these fields, simply fire an action event on the Compare button. The event-handling code for the Compare button will then handle the file comparison.

```
// A JavaFX version of the file comparison program shown in
// Try This 16-1.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import java.io.*;

public class JavaFXFileComp extends Application {

  TextField tfFirst;  // holds the first file name
  TextField tfSecond; // holds the second file name

  Button btnComp; // button to compare the files

  Label labFirst, labSecond; // displays prompts
  Label labResult; // displays results and error messages

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Compare Files");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);
```

```
// Center the controls in the scene.
rootNode.setAlignment(Pos.CENTER);

// Create a scene.
Scene myScene = new Scene(rootNode, 180, 180);

// Set the scene on the stage.
myStage.setScene(myScene);

// Create the text fields for the file names.
tfFirst = new TextField();
tfSecond = new TextField();

// Set preferred column sizes.
tfFirst.setPrefColumnCount(12);
tfSecond.setPrefColumnCount(12);

// Set prompts for file names.
tfFirst.setPromptText("Enter file name.");
tfSecond.setPromptText("Enter file name.");

// Create the Compare button.
btnComp = new Button("Compare");

// Create the labels.
labFirst = new Label("First file: ");
labSecond = new Label("Second file: ");
labResult = new Label("");

// Use lambda expressions to handle action events for the
// text fields. These handlers simply fire the Compare button.
tfFirst.setOnAction( (ae) -> btnComp.fire());
tfSecond.setOnAction( (ae) -> btnComp.fire());

// Handle the action event for the Compare button.
btnComp.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    int i=0, j=0;

    // First, confirm that both file names have
    // been entered.
    if(tfFirst.getText().equals("")) {
      labResult.setText("First file name missing.");
      return;
    }
    if(tfSecond.getText().equals("")) {
      labResult.setText("Second file name missing.");
      return;
    }

      // Compare files. Use try-with-resources to manage the files.
      try (FileInputStream f1 = new FileInputStream(tfFirst.getText());
           FileInputStream f2 = new FileInputStream(tfSecond.getText()))
      {
        // Check the contents of each file.
        do {
          i = f1.read();
          j = f2.read();
          if(i != j) break;
        } while(i != -1 && j != -1);

        if(i != j)
          labResult.setText("Files are not the same.");
        else
          labResult.setText("Files compare equal.");

      } catch(IOException exc) {
        labResult.setText("File Error Encountered");
      }
    }
  });

  // Add controls to the scene graph.
  rootNode.getChildren().addAll(labFirst, tfFirst, labSecond, tfSecond,
                                btnComp, labResult);

  // Show the stage and its scene.
  myStage.show();
 }
}
```

14. Modify the **EffectsAndTransformsDemo** program so the Rotate button is also blurred. Use a blur width and height of 5 and an iteration count of 2.

To add blur to the Rotate button, first create the **BoxBlur** instance like this:

```
BoxBlur rotateBlur = new BoxBlur(5.0, 5.0, 2);
```

Then add the following line:

```
btnRotate.setEffect(rotateBlur);
```

After making these changes, the Rotate button will be blurred and can also be rotated.

PREV
Chapter 17: Introducing JavaFX

NEXT
Appendix B: Using Java's Documentation Comments