Module 12

Enumerations, Autoboxing, and Static Import

CRITICAL SKILLS

12.1	Understand enumeration fundamentals
12.2	Use the class-based features of enumerations
12.3	Apply the **values( )** and **valueof( )** methods to enumerations
12.4	Create enumerations that have constructors, instance variables, and methods
12.5	Employ the **ordinal( )** and **compareTo( )** methods that enumerations inherit from **Enum**
12.6	Use Java's type wrappers
12.7	Know the basics of autoboxing and auto-unboxing
12.8	Use autoboxing with methods
12.9	Understand how autoboxing works with expressions
12.10	Apply static import
12.11	Gain an overview of metadata

With the release of J2SE 5 in late 2004, Java was substantially expanded by the addition of several new language features. These additions fundamentally alter the character and scope of the language. The features added by J2SE 5 are shown here:

- Generics
- Enumerations
- Autoboxing/unboxing
- The enhanced **for** loop
- Variable-length arguments (varargs)
- Static import
- Metadata (annotations)

Features such as enumerations and autoboxing/unboxing answer long-standing needs. Others, such as generics and metadata, broke new ground. In both cases, these new features have profoundly changed Java.

Two of these new features, the enhanced **for** loop and varargs, have already been discussed. This module examines in detail enumerations, autoboxing, and static import. An overview of metadata ends the module. Module 13 discusses generics.

If you are using an older version of Java that predates the J2SE 5 release, you will not be able to use the features described here and in Module 13.

CRITICAL SKILL 12.1 Enumerations

The enumeration is a common programming feature that is found in many other computer languages. However, it was not part of the original specification for Java. One reason for this is that the enumeration is technically a convenience, rather than a necessity. However, over the years, many programmers have wanted Java to support enumerations because they offer an elegant, structured solution to a variety of programming tasks. This request was granted by the release of J2SE 5, which added enumerations to Java.

In its simplest form, an *enumeration* is a list of named constants that define a new data type. An object of an enumeration type can hold only the values that are defined by the list. Thus, an enumeration gives you a way to precisely define a new type of data that has a fixed number of valid values.

Enumerations are common in everyday life. For example, an enumeration of the coins used in the United States is penny, nickel, dime, quarter, half-dollar, and dollar. An enumeration of the months in the year consists of the names January through December. An enumeration of the days of the week is Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday.

From a programming perspective, enumerations are useful whenever you need to define a set of values that represent a collection of items. For example, you might use an enumeration to represent a set of status codes, such as success, waiting, failed, and retrying, which indicate the progress of some action. In the past, such values were defined as **final** variables, but enumerations offer a much more structured approach.

Enumeration Fundamentals

An enumeration is created using the new **enum** keyword. For example, here is a simple enumeration that lists various forms of transportation:

```
// An enumeration of transportation.
enum Transport {
  CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

The identifiers **CAR**, **TRUCK**, and so on, are called *enumeration constants*. Each is implicitly declared as a public, static member of **Transport**. Furthermore, the enumeration constants' type is the type of the enumeration in which the constants are declared, which is **Transport** in this case. Thus, in the language of Java, these constants are called *self-typed*, where "self" refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way that you do one of the primitive types. For example, this declares **tp** as a variable of enumeration type **Transport**:

```
Transport tp;
```

Because **tp** is of type **Transport**, the only values that it can be assigned are those defined by the enumeration. For example, this assigns **tp** the value **AIRPLANE**:

```
tp = Transport.AIRPLANE;
```

Notice that the symbol **AIRPLANE** is qualified by **Transport**.

Two enumeration constants can be compared for equality by using the = = relational operator. For example, this statement compares the value in **tp** with the **TRAIN** constant:

```
if(tp == Transport.TRAIN) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(tp) {
  case CAR:
    // ...
  case TRUCK:
    // ...
```

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, **TRUCK**, not **Transport.TRUCK**, is used. This is because the type of the enumeration in the **switch** expression has already implicitly specified the **enum** type of the **case** constants. There is no need to qualify the constants in the **case** statements with their **enum** type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a **println( )** statement, its name is output. For example, given this statement:

```
System.out.println(Transport.BOAT);
```

the name **BOAT** is displayed.

The following program puts together all of the pieces and demonstrates the **Transport** enumeration.

```
// An enumeration of Transport varieties.
enum Transport {
  CAR, TRUCK, AIRPLANE, TRAIN, BOAT  <-- Declare an enumeration.
}

class EnumDemo {
  public static void main(String args[])
  {
    Transport tp;  <-- Declare a Transport reference.

    tp = Transport.AIRPLANE;  <-- Assign tp the constant AIRPLANE.

    // Output an enum value.
```

```
    System.out.println("Value of tp: " + tp);
    System.out.println();

    tp = Transport.TRAIN;

    // Compare two enum values.
    if(tp == Transport.TRAIN)
      System.out.println("tp contains TRAIN.\n");

    // Use an enum to control a switch statement.
    switch(tp) {
      case CAR:
        System.out.println("A car carries people.");
        break;
      case TRUCK:
        System.out.println("A truck carries freight.");
        break;
      case AIRPLANE:
        System.out.println("An airplane flies.");
        break;
      case TRAIN:
        System.out.println("A train runs on rails.");
        break;
      case BOAT:
        System.out.println("A boat sails on water.");
        break;
    }
  }
}
```

Compare two **Transport** objects for equality.

Use an enumeration to control a **switch** statement.

The output from the program is shown here:

```
Value of tp: AIRPLANE

tp contains TRAIN.

A train runs on rails.
```

Before moving on, it's necessary to make one stylistic point. The constants in **Transport** use uppercase. (Thus, **CAR**, not **car**, is used.) However, the use of uppercase is not required. In other words, there is no rule that requires enumeration constants to be in uppercase. Because enumerations often replace **final** variables, which have traditionally used uppercase, some programmers believe that uppercasing enumeration constants is also appropriate. There are, of course, other viewpoints and styles. The examples in this book will use uppercase for enumeration constants, for consistency.

Progress Check

1. An enumeration defines a list of _____ constants.
2. What keyword declares an enumeration?
3. Given

```
enum Directions {
  LEFT, RIGHT, UP, DOWN
}
```

What is the data type of **UP**?

CRITICAL SKILL 12.2

Java Enumerations Are Class Types

Although the preceding examples show the mechanics of creating and using an enumeration, they don't show all of its capabilities. Unlike the way enumerations are implemented in many other languages, *Java implements enumerations as class types*. Although you don't instantiate an **enum** using **new**, it otherwise acts much like other classes. The fact that **enum** defines a class enables the Java enumeration to have powers that enumerations in other languages do not. For example, you can give it constructors, add instance variables and methods, and even implement interfaces.

CRITICAL SKILL 12.3

The values( ) and valueOf( ) Methods

All enumerations automatically have two predefined methods: **values( )** and **valueOf( )**. Their general forms are shown here:

public static *enum-type*[ ] values( )

public static *enum-type* valueOf(String *str*)

The **values( )** method returns an array that contains a list of the enumeration constants. The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For

1. named
2. **enum**
3. The data type of **UP** is **Directions** because enumerated constants are self-typed.

example, in the case of the **Transport** enumeration shown earlier, the return type of **Transport.valueOf("TRAIN")** is **Transport**. The value returned is **TRAIN**.

The following program demonstrates the **values( )** and **valueOf( )** methods.

```
// Use the built-in enumeration methods.

// An enumeration of Transport varieties.
enum Transport {
  CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Transport tp;

    System.out.println("Here are all Transport constants");

    // use values()
    Transport allTransports[] = Transport.values();
    for(Transport t : allTransports)
      System.out.println(t);

    System.out.println();

    // use valueOf()
    tp = Transport.valueOf("AIRPLANE");
    System.out.println("tp contains " + tp);

  }
}
```

Obtain an array of **Transport** constants.

Obtain the constant with the name **AIRPLANE**.

The output from the program is shown here:

```
Here are all Transport constants
CAR
TRUCK
AIRPLANE
TRAIN
BOAT

tp contains AIRPLANE
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values( )**. For the sake of illustration, the variable **allTransports**

was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allTransports** variable:

```
for(Transport t : Transport.values())
  System.out.println(t);
```

Now, notice how the value corresponding to the name **AIRPLANE** was obtained by calling **valueOf( )**:

```
tp = Transport.valueOf("AIRPLANE");
```

As explained, **valueOf( )** returns the enumeration value associated with the name of the constant represented as a string.

CRITICAL SKILL 12.4

Constructors, Methods, Instance Variables, and Enumerations

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, an enumeration can define constructors, add methods, and have instance variables. When you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Each enumeration constant can call any method defined by the enumeration. Each enumeration constant has its own copy of any instance variables defined by the enumeration. The following version of **Transport** illustrates the use of a constructor, an instance variable, and a method. It gives each type of transportation a typical speed.

```
// Use an enum constructor, instance variable, and method.
enum Transport {
  CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22);

  private int speed; // typical speed of each transport

  // Constructor
  Transport(int s) { speed = s; }

  int getSpeed() { return speed; }
}

class EnumDemo3 {
  public static void main(String args[])
  {
    Transport tp;
```

Notice the initialization values.

Add an instance variable.

Add a constructor.

Add a method.

```
    // Display speed of an airplane.
    System.out.println("Typical speed for an airplane is " +
                       Transport.AIRPLANE.getSpeed() +
                       " miles per hour.\n");

    // Display all Transports and speeds.
    System.out.println("All Transport speeds: ");
    for(Transport t : Transport.values())
      System.out.println(t + " typical speed is " +
                         t.getSpeed() +
                         " miles per hour.");
  }
}
```

Obtain the speed by calling **getSpeed( )**.

The output is shown here:

```
Typical speed for an airplane is 600 miles per hour.

All Transport speeds:
CAR typical speed is 65 miles per hour.
TRUCK typical speed is 55 miles per hour.
AIRPLANE typical speed is 600 miles per hour.
TRAIN typical speed is 70 miles per hour.
BOAT typical speed is 22 miles per hour.
```

This version of **Transport** adds three things. The first is the instance variable **speed**, which is used to hold the speed of each kind of transport. The second is the **Transport** constructor, which is passed the speed of a transport. The third is the method **getSpeed( )**, which returns the value of **Speed**.

When the variable **tp** is declared in **main( )**, the constructor for **Transport** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses, after each constant, as shown here:

```
CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22);
```

These values are passed to the **s** parameter of **Transport( )**, which then assigns this value to **speed**. The constructor is called once for each constant. There is something else to notice about the list of enumeration constants: it is terminated by a semicolon. That is, the last constant, **BOAT**, is followed by semicolon. When an enumeration contains other members, the enumeration list must end in a semicolon.

Because each enumeration constant has its own copy of **speed**, you can obtain the speed of a specified type of transport by calling **getSpeed( )**. For example, in **main( )** the speed of an airplane is obtained by the following call:

```
Transport.AIRPLANE.getSpeed()
```

Ask the Expert

Q: **Now that enumerations are part of Java, should I avoid the use of** final **variables?**

A: No. Enumerations are appropriate when you are working with lists of items that must be represented by identifiers. A **final** variable is appropriate when you have a constant value, such as an array size, that will be used in many places. Thus, each has its own use. The advantage of enumerations is that now final variables don't have to be pressed into service for a job for which they are not ideally suited.

The speed of each transport is obtained by cycling through the enumeration using a **for** loop. Because there is a copy of **speed** for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is available only when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class.

Two Important Restrictions

There are two restrictions that apply to enumerations. First, an enumeration can't inherit another class. Second, an **enum** cannot be a superclass. This means that an **enum** can't be extended. Otherwise, **enum** acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

CRITICAL SKILL 12.5

Enumerations Inherit Enum

Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. Most often you won't need to use these methods, but there are two that you may occasionally employ: **ordinal( )** and **compareTo( )**.

The **ordinal( )** method obtains a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value*. The **ordinal( )** method is shown here:

```
final int ordinal( )
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the **Transport** enumeration, **CAR** has an ordinal value of zero, **TRUCK** has an ordinal value of 1, **AIRPLANE** has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method. It has this general form:

final int compareTo(*enum-type e*)

Here, *enum-type* is the type of the enumeration and *e* is the constant being compared to the invoking constant. Remember, both the invoking constant and *e* must be of the same enumeration. If the invoking constant has an ordinal value less than *e*'s, then **compareTo( )** returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than *e*'s, then a positive value is returned.

The following program demonstrates **ordinal( )** and **compareTo( )**.

```
// Demonstrate ordinal() and compareTo().

// An enumeration of Transport varieties.
enum Transport {
  CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Transport tp, tp2, tp3;

    // Obtain all ordinal values using ordinal().
    System.out.println("Here are all Transport constants" +
                       " and their ordinal values: ");
    for(Transport t : Transport.values())
      System.out.println(t + " " + t.ordinal()); <-- Obtain ordinal values.

    tp =  Transport.AIRPLANE;
    tp2 = Transport.TRAIN;
    tp3 = Transport.AIRPLANE;

    System.out.println();

                                    Compare ordinal values.
    // Demonstrate compareTo()      |
    if(tp.compareTo(tp2) < 0) <-----
      System.out.println(tp + " comes before " + tp2);

    if(tp.compareTo(tp2) > 0)
```

```
      System.out.println(tp2 + " comes before " + tp);

    if(tp.compareTo(tp3) == 0)
      System.out.println(tp + " equals " + tp3);
  }
}
```

The output from the program is shown here:

```
Here are all Transport constants and their ordinal values:
CAR 0
TRUCK 1
AIRPLANE 2
TRAIN 3
BOAT 4

AIRPLANE comes before TRAIN
AIRPLANE equals AIRPLANE
```

Progress Check

1. What does **values( )** return?
2. Can an enumeration have a constructor?
3. What is the ordinal value of an enumeration constant?

Project 12-1 A Computer-Controlled Traffic Light

`TrafficLightDemo.java`

Enumerations are particularly useful when your program needs a set of constants, but the actual values of the constants are arbitrary, as long as all differ. This type of situation comes up quite often when programming. One common instance involves handling the states in which some device can exist. For example, imagine that you are writing a program that controls a traffic light. Your traffic light code must automatically cycle through the light's three states: green, yellow, and red. It also must enable other code to know the current color of the light and let the color of the light be

1. The **values( )** method returns an array that contains a list of all the constants defined by the invoking enumeration.
2. Yes.
3. The ordinal value of an enumeration constant describes its position in the list of constants, with the first constant having the ordinal value of zero.

set to a known initial value. This means that the three states must be represented in some way. Although it would be possible to represent these three states by integer values (for example, the values 1, 2, and 3) or by strings (such as "red", "green", and "yellow"), an enumeration offers a much better approach. Using an enumeration results in code that is more efficient than if strings represented the states and more structured than if integers represented the states.

In this project, you will create a simulation of an automated traffic light, as just described. This project not only demonstrates an enumeration in action, it also shows another example of multithreading and synchronization.

Step by Step

1. Create a file called **TrafficLightDemo.java**.
2. Begin by defining an enumeration called **TrafficLightColor** that represents the three states of the light, as shown here:

```
// An enumeration of the colors of a traffic light.
enum TrafficLightColor {
  RED, GREEN, YELLOW
}
```

   Whenever the color of the light is needed, its enumeration value is used.

3. Next, begin defining **TrafficLightSimulator**, as shown next. **TrafficLightSimulator** is the class that encapsulates the traffic light simulation.

```
// 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

  TrafficLightSimulator(TrafficLightColor init) {
    tlc = init;

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

  TrafficLightSimulator() {
    tlc = TrafficLightColor.RED;

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

(continued)

Notice that **TrafficLightSimulator** implements **Runnable**. This is necessary because a separate thread is used to run each traffic light. This thread will cycle through the colors. Two constructors are created. The first lets you specify the initial light color. The second defaults to red. Both start a new thread to run the light.

Now look at the instance variables. A reference to the traffic light thread is stored in **thrd**. The current traffic light color is stored in **tlc**. The **stop** variable is used to stop the simulation. It is initially set to false. The light will run until this variable is set to true.

4. Next, add the **run( )** method, shown here, which begins running the traffic light.

```
// Start up the light.
public void run() {
  while(!stop) {

    try {
      switch(tlc) {
        case GREEN:
          Thread.sleep(10000); // green for 10 seconds
          break;
        case YELLOW:
          Thread.sleep(2000);  // yellow for 2 seconds
          break;
        case RED:
          Thread.sleep(12000); // red for 12 seconds
          break;
      }
    } catch(InterruptedException exc) {
      System.out.println(exc);
    }
    changeColor();
  }
}
```

This method cycles the light through the colors. First, it sleeps an appropriate amount of time, based on the current color. Then, it calls **changeColor( )** to change to the next color in the sequence.

5. Now, add the **changeColor( )** method, as shown here:

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

```
      tlc = TrafficLightColor.RED;
      break;
    case GREEN:
     tlc = TrafficLightColor.YELLOW;
  }

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

The **switch** statement examines the color currently stored in **tlc** and then assigns the next color in the sequence. Notice that this method is synchronized. This is necessary because it calls **notify( )** to signal that a color change has taken place. (Recall that **notify( )** can be called only from a synchronized method.)

6. The next method is **waitForChange( )**, which waits until the color of the light is changed.

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

This method simply calls **wait( )**. This call won't return until **changeColor( )** executes a call to **notify( )**. Thus, **waitForChange( )** won't return until the color has changed.

7. Finally, add the methods **getColor( )**, which returns the current light color, and **cancel( )**, which stops the traffic light thread by setting **stop** to true. These methods are shown here:

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

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

8. Here is all the code assembled into a complete program that demonstrates the traffic light:

```
// A simulation of a traffic light that uses
// an enumeration to describe the light's color.

// An enumeration of the colors of a traffic light.
```

(continued)

```
enum TrafficLightColor {
  RED, GREEN, YELLOW
}

// 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

  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) {

      try {
        switch(tlc) {
          case GREEN:
            Thread.sleep(10000); // green for 10 seconds
            break;
          case YELLOW:
            Thread.sleep(2000);  // yellow for 2 seconds
            break;
          case RED:
            Thread.sleep(12000); // red for 12 seconds
            break;
        }
      } 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;
    }

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

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

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

  // Stop the traffic light.
  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();
    }
```

(continued)

```
    tl.cancel();
  }
}
```

The following output is produced. As you can see, the traffic light cycles through the colors in order of green, yellow, and red:

```
GREEN
YELLOW
RED
GREEN
YELLOW
RED
GREEN
YELLOW
RED
```

In the program, notice how the use of the enumeration simplifies and adds structure to the code that needs to know the state of the traffic light. Because the light can have only three states (red, green, or yellow), the use of an enumeration ensures that only these values are valid, thus preventing accidental misuse.

9. It is possible to improve the preceding program by taking advantage of the class capabilities of an enumeration. For example, by adding a constructor, instance variable, and method to **TrafficLightColor**, you can substantially improve the preceding programming. This improvement is left as an exercise. See Mastery Check, question 4.

Autoboxing

With the release of J2SE 5, Java has added two features that were long desired by Java programmers: *autoboxing* and *auto-unboxing*. Autoboxing/unboxing greatly simplifies and streamlines code that must convert primitive types into objects, and vice versa. Because such situations are found frequently in Java code, the benefits of autoboxing/unboxing affect nearly all Java programmers. As you will see in Module 13, autoboxing/unboxing contributes greatly to the usability of another new feature: generics. The addition of autoboxing/unboxing subtly changes the relationship between objects and the primitive types. These changes are more profound than the conceptual simplicity of autoboxing/unboxing might at first suggest. Their effects are widely felt throughout the Java language.

Autoboxing/unboxing is directly related to Java's type wrappers, and to the way that values are moved into and out of an instance of a wrapper. For this reason, we will begin with an overview of the type wrappers and the process of manually boxing and unboxing values.

CRITICAL SKILL 12.6

Type Wrappers

As you know, Java uses primitive types, such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these basic types would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object. The type wrapper classes were introduced briefly in Module 10. Here, we will look at them more closely.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**, which are packaged in **java.lang**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different numeric types. These methods are shown here:

```
byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )
```

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer** and **Double**:

Integer(int *num*)

Integer(String *str*)

Double(double *num*)

Double(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.

The process of encapsulating a value within an object is called *boxing*. Prior to J2SE 5, all boxing took place manually, with the programmer explicitly constructing an instance of a wrapper with the desired value. For example, this line manually boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

In this example, a new **Integer** object with the value 100 is explicitly created and a reference to this object is assigned to **iOb**.

The process of extracting a value from a type wrapper is called *unboxing*. Again, prior to J2SE 5, all unboxing also took place manually, with the programmer explicitly calling a method on the wrapper to obtain its value. For example, this manually unboxes the value in **iOb** into an **int**.

```
int i = iOb.intValue();
```

Here, **intValue( )** returns the value encapsulated within **iOb** as an **int**.

The following program demonstrates the preceding concepts.

```
// Demonstrate manual boxing and unboxing with a type wrapper.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = new Integer(100); <-- Manually box the value 100.

    int i = iOb.intValue(); <-- Manually unbox the value in iOb.
```

```
    System.out.println(i + " " + iOb); // displays 100 100
  }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**. Finally, it displays the values of **i** and **iOb**, both of which are 100.

The same general procedure used by the preceding example to manually box and unbox values has been employed since the original version of Java. Although this approach to boxing and unboxing works, it is both tedious and error-prone because it requires the programmer to manually create the appropriate object to wrap a value and to explicitly obtain the proper primitive type when its value is needed. Fortunately, J2SE 5 fundamentally improves on these essential procedures with the addition of autoboxing/unboxing.

CRITICAL SKILL 12.7

Autoboxing Fundamentals

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue( )**.

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that no object is explicitly created through the use of **new**. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

The following program demonstrates the preceding statements.

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
  public static void main(String args[]) {

    Integer iOb = 100; // autobox an int

    int i = iOb; // auto-unbox

    System.out.println(i + " " + iOb);  // displays 100 100
  }
}
```

Autobox and then auto-unbox the value 100.

Progress Check

1. What is the type wrapper for **double**?
2. When you box a primitive value, what happens?
3. Autoboxing is the feature that automatically boxes a primitive value into an object of its corresponding type wrapper. True or False?

CRITICAL SKILL 12.8

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object, and auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method or when a value is returned by a method. For example, consider the following:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
  // This method has an Integer parameter.
  static void m(Integer v) {
    System.out.println("m() received " + v);
  }
```

Receives an **Integer**.

1. **Double**
2. When a primitive value is boxed, its value is placed inside an object of its corresponding type wrapper.
3. True.

```
  // This method returns an int.
  static int m2() {
    return 10;
  }

  // This method returns an Integer.
  static Integer m3() {
    return 99; // autoboxing 99 into an Integer.
  }

  public static void main(String args[]) {

    // Pass an int to m().  Because m() has an Integer
    // parameter, the int value passed is automatically boxed.
    m(199);

    // Here, iOb receives the int value returned by m2().
    // This value is automatically boxed so that it can be
    // assigned to iOb.
    Integer iOb = m2();
    System.out.println("Return value from m2() is " + iOb);

    // Next, m3() is called. It returns an Integer value
    // which is auto-unboxed into an int.
    int i = m3();
    System.out.println("Return value from m3() is " + i);

    // Next, Math.sqrt() is called with iOb as an argument.
    // In this case, iOb is auto-unboxed and its value promoted to
    // double, which is the type needed by sqrt().
    iOb = 100;
    System.out.println("Square root of iOb is " + Math.sqrt(iOb));
  }
}
```

Returns an **int**.

Returns an **Integer**.

This program displays the following result:

```
m() received 199
Return value from m2() is 10
Return value from m3() is 99
Square root of iOb is 10.0
```

In the program, notice that **m( )** specifies an **Integer** parameter. Inside **main( )**, **m( )** is passed the **int** value 199. Because **m( )** is expecting an **Integer**, this value is automatically boxed. Next, **m2( )** is called. It returns the **int** value 10. This **int** value is assigned to **iOb** in **main( )**. Because **iOb** is an **Integer**, the value returned by **m2( )** is autoboxed. Next, **m3( )** is

called. It returns an **Integer** that is auto-unboxed into an **int**. Finally, **Math.sqrt( )** is called with **iOb** as an argument. In this case, **iOb** is auto-unboxed and its value promoted to **double**, since that is the type expected by **Math.sqrt( )**.

CRITICAL SKILL 12.9

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program.

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 99;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, its value is increased by 10,
    // and the result is boxed and stored back in iOb.
    iOb += 10;
    System.out.println("After iOb += 10: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
```

Autoboxing/ unboxing occurs in expressions.

```
    System.out.println("i after expression: " + i);
  }
}
```

The output is shown here:

```
Original value of iOb: 99
After ++iOb: 100
After iOb += 10: 110
iOb2 after expression: 146
i after expression: 146
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Because of auto-unboxing, you can use integer numeric objects, such as an **Integer**, to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;

switch(iOb) {
  case 1: System.out.println("one");
    break;
  case 2: System.out.println("two");
    break;
  default: System.out.println("error");
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the past, such code would have involved casts and calls to methods such as **intValue( )**.

A Word of Warning

Now that Java includes autoboxing and auto-unboxing, one might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!
Double a, b, c;
```

```
a = 10.2;
b = 11.4;
c = 9.8;

Double avg = (a + b + c) / 3;
```

In this example, objects of type **Double** hold values, which are then averaged and the result assigned to another **Double** object. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a "back door" way of eliminating the primitive types.

Progress Check

1. Will a primitive value be autoboxed when it is passed as an argument to a method that is expecting a type wrapper object?
2. Because of the limits imposed by the Java run-time system, autoboxing/unboxing will not occur on objects used in expressions. True or False?
3. Because of autoboxing/unboxing, you should use objects rather than primitive types for performing most arithmetic operations. True or False?

CRITICAL SKILL 12.10

Static Import

J2SE 5 expanded the use of the **import** keyword so that it supports a new feature called *static import*. By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.

1. Yes.
2. False.
3. False.

To understand the usefulness of static import, let's begin with an example that *does not* use it. The following program computes the solutions to a quadratic equation, which has this form:

$ax^2 + bx + c = 0$

The program uses two static methods from Java's built-in math class **Math**, which is part of **java.lang**. The first is **Math.pow( )**, which returns a value raised to a specified power. The second is **Math.sqrt( )**, which returns the square root of its argument.

```
// Find the solutions to a quadratic equation.
class Quadratic {
  public static void main(String args[]) {

    // a, b, and c represent the coefficients in the
    // quadratic equation: ax^2 + bx + c = 0
    double a, b, c, x;

    // Solve 4x^2 + x - 3 = 0 for x.
    a = 4;
    b = 1;
    c = -3;

    // Find first solution.
    x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("First solution: " + x);

    // Find second solution.
    x = (-b - Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("Second solution: " + x);
  }
}
```

Because **pow( )** and **sqrt( )** are static methods, they must be called through the use of their class' name, **Math**. This results in a somewhat unwieldy expression:

```
x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
```

Furthermore, having to specify the class name each time **pow( )** or **sqrt( )** (or any of Java's other math methods, such as **sin( )**, **cos( )**, and **tan( )**) are used, can become tedious.

You can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program.

```
// Use static import to bring sqrt() and pow() into view.

import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

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

    // a, b, and c represent the coefficients in the
    // quadratic equation: ax^2 + bx + c = 0
    double a, b, c, x;

    // Solve 4x^2 + x - 3 = 0 for x.
    a = 4;
    b = 1;
    c = -3;

    // Find first solution.
    x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("First solution: " + x);

    // Find second solution.
    x = (-b - sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("Second solution: " + x);
  }
}
```

Use static import to bring **sqrt( )** and **pow( )** into view.

In this version, the names **sqrt** and **pow** are brought into view by these static import statements:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

After these statements, it is no longer necessary to qualify **sqrt( )** or **pow( )** with its class name. Therefore, the expression can more conveniently be specified, as shown here:

```
x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
```

As you can see, this form is considerably shorter and easier to read.

There are two general forms of the **import static** statement. The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

import static *pkg.type-name.static-member-name*;

Here, *type-name* is the name of a class or interface that contains the desired static member. Its full package name is specified by *pkg*. The name of the member is specified by *static-member-name*.

The second form of static import imports all static members. Its general form is shown here:

import static *pkg.type-name*.*;

If you will be using many static methods or fields defined by a class, then this form lets you bring them into view without having to specify each individually. Therefore, the preceding program could have used this single **import** statement to bring both **pow( )** and **sqrt( )** (and *all other* static members of **Math**) into view:

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

Of course, static import is not limited just to the **Math** class or just to methods. For example, this brings the static field **System.out** into view:

```
import static java.lang.System.out;
```

After this statement, you can output to the console without having to qualify **out** with **System**, as shown here:

```
out.println("After importing System.out, you can use out directly.");
```

Whether importing **System.out** as just shown is a good idea is subject to debate. Although it does shorten the statement, it is no longer instantly clear to anyone reading the program that the **out** being referred to is **System.out**.

As convenient as static import can be, it is important not to abuse it. Remember, the reason that Java organizes its libraries into packages is to avoid namespace collisions. When you import static members, you are bringing those members into the global namespace. Thus, you are increasing the potential for namespace conflicts and the inadvertent hiding of other names. If you are using a static member once or twice in the program, it's best not to import it. Also, some static names, such as **System.out**, are so recognizable that you might not want to import them. Static import is designed for those situations in which you are using a static member repeatedly, such as when performing a series of mathematical computations. In essence, you should use, but not abuse, this feature.

Ask the Expert

Q: **Using static import, can I import the static members of classes that I create?**

A: Yes, you can use static import to import the static members of classes and interfaces you create. Doing so is especially convenient when you define several static members that are used frequently throughout a large program. For example, if a class defines a number of **static final** constants that define various limits, then using static import to bring them into view will save you a lot of tedious typing.

CRITICAL SKILL 12.11 Metadata

Of the new features added to Java by J2SE 5, metadata is the most innovative. This powerful new facility enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. However, this information can be used by various tools, during both development and deployment. For example, an annotation might be processed by a source-code generator, by the compiler, or by a deployment tool. Although Sun refers to this feature as metadata, the term *program annotation facility* is also used and is probably more descriptive.

Metadata is a large and sophisticated topic, and it is far beyond scope of this book to cover it in detail. However, a brief overview is given here so that you will be familiar with the concept.

A detailed discussion of metadata and annotations can be found in my book *Java: The Complete Reference, J2SE 5 Edition* (McGraw-Hill/Osborne, 2005).

Metadata is created through a mechanism based on the **interface**. Here is a simple example:

```
// A simple annotation type.
@interface MyAnno {
  String str();
  int val();
}
```

This declares an annotation called **MyAnno**. Notice the **@** that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice

the two members **str( )** and **val( )**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields.

All annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package.

Once you have declared an annotation, you can use it to annotate a declaration. Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method:

```
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ...
```

This annotation is linked with the method **myMeth( )**. Look closely at the annotation syntax. The name of the annotation, preceded by an **@**, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**. *Notice that no parentheses follow* ***str*** *in this assignment. When an annotation member is given* a value, only its name is used. Thus, annotation members look like fields in this context.

Annotations that don't have parameters are called *marker annotations*. These are specified without passing any arguments and without using parentheses. Their sole purpose is to mark a declaration with some attribute.

At the time of this writing, Java defines seven built-in annotations. Four are imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**, and **@Inherited**. Three, **@Override**, **@Deprecated**, and **@SuppressWarnings**, are included in **java.lang**. The built-in annotations are shown in Table 12-1.

Here is an example that uses **@Deprecated** to mark the **MyClass** class and the **getMsg( )** method. When you try to compile this program, warnings will report the use of these deprecated elements.

```
// An example that uses @Deprecated.

// Deprecate a class.
@Deprecated
class MyClass {
  private String msg;

  MyClass(String m) {
    msg = m;
```

Mark a class as deprecated.

```
  }

  // Deprecate a method within a class.
  @Deprecated
  String getMsg() {
    return msg;
  }

  // ...
}

class AnnoDemo {
  public static void main(String args[]) {
    MyClass myObj = new MyClass("test");

    System.out.println(myObj.getMsg());
  }
}
```

Mark a method as deprecated.

Annotation	Description
@Retention	Specifies the retention policy that will be associated with the annotation. The retention policy determines how long an annotation is present during the compilation and deployment process.
@Documented	A marker annotation that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.
@Target	Specifies the types of declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which must be a constant from the **ElementType** enumeration, which defines various constants, such as **CONSTRUCTOR**, **FIELD**, and **METHOD**. The argument determines the types of declarations to which the annotation can be applied.
@Inherited	A marker annotation that causes the annotation for a superclass to be inherited by a subclass.
@Override	A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded. This is a marker annotation.
@Deprecated	A marker annotation that indicates that a declaration is obsolete and has been replaced by a newer form.
@SuppressWarnings	Specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form.

Table 12-1 The Built-in Annotations

Progress Check

1. Show the two forms of static import.
2. Show how to import **Thread**'s **sleep( )** method so that it can be used without being qualified by **Thread**.
3. Static import works with methods, but not variables. True or False?
4. An annotation begins with a/an _____.

Module 12 Mastery Check

1. Enumeration constants are said to be *self-typed.* What does this mean?
2. What class do all enumerations automatically inherit?
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
}
```

4. The traffic light simulation developed in Project 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.)
5. Define boxing and unboxing. How does autoboxing/unboxing affect these actions?

---

1. import static *pkg.type-name.static-member-name*;
   import static *pkg.type-name.**;
2. import static java.lang.Thread.sleep;
3. False.
4. @