```
if(n < 0) {
   System.out.println("n is negative!");
   return; // or throw an exception
}
```

With **assert**, you need only one line of code. Furthermore, you don't have to remove the **assert** statements from your released code.

Assertion Enabling and Disabling Options

When executing code, you can disable assertions by using the **-da** option. You can enable or disable a specific package by specifying its name after the **-ea** or **-da** option. For example, to enable assertions in a package called **MyPack**, use

```
-ea:MyPack
```

To disable assertions in **MyPack**, use

```
-da:MyPack
```

To enable or disable all subpackages of a package, follow the package name with three dots. For example,

```
-ea:MyPack...
```

You can also specify a class with the **-ea** or **-da** option. For example, this enables **AssertDemo** individually:

```
-ea:AssertDemo
```

Static Import

JDK 5 added a new feature to Java called *static import* that expands the capabilities of the **import** keyword. 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.

To understand the usefulness of static import, let's begin with an example that does *not* use it. The following program computes the hypotenuse of a right triangle. It 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.

```
// Compute the hypotenuse of a right triangle.
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;
```

```
    side1 = 3.0;
    side2 = 4.0;

    // Notice how sqrt() and pow() must be qualified by
    // their class name, which is Math.
    hypot = Math.sqrt(Math.pow(side1, 2) +
                      Math.pow(side2, 2));

    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

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 hypotenuse calculation:

```
hypot = Math.sqrt(Math.pow(side1, 2) +
                  Math.pow(side2, 2));
```

As this simple example illustrates, 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( )**) is used can grow 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;

// Compute the hypotenuse of a right triangle.
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Here, sqrt() and pow() can be called by themselves,
    // without their class name.
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));

    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

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 their class name. Therefore, the hypotenuse calculation can more conveniently be specified, as shown here:

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

As you can see, this form is considerably more readable.

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 of a given class or interface. 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.**

One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.

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 for 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.

Invoking Overloaded Constructors Through this( )

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the **this** keyword. The general form is shown here:

this(*arg-list*)

When **this( )** is executed, the overloaded constructor that matches the parameter list specified by *arg-list* is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to **this( )** must be the first statement within the constructor.

To understand how **this( )** can be used, let's work through a short example. First, consider the following class that *does not* use **this( )**:

```
class MyClass {
  int a;
  int b;

  // initialize a and b individually
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // initialize a and b to the same value
  MyClass(int i) {
    a = i;
    b = i;
  }

  // give a and b default values of 0
  MyClass( ) {
    a = 0;
    b = 0;
  }
}
```