Wrapper Classes

Each of Java's eight primitive data types has a class dedicated to it. These are known as *wrapper classes*, because they "wrap" the primitive data type into an object of that class. So, there is an `Integer` class that holds an `int` variable, there is a `Double` class that holds a `double` variable, and so on. The wrapper classes are part of the `java.lang` package, which is imported by default into all Java programs. The following discussion focuses on the `Integer` wrapper class, but applies in a general sense to all eight wrapper classes. Consult the Java API documentation for more details.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int     x = 25;
Integer y = new Integer(33);
```

The first statement declares an `int` variable named `x` and initializes it with the value 25. The second statement instantiates an `Integer` object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable `y`. The memory assignments from these two statements are visualized in Figure 1.

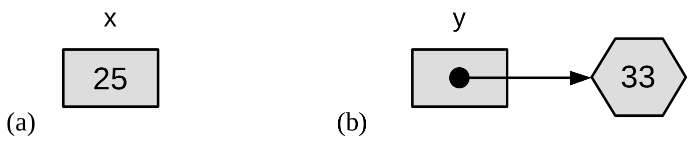

Figure 1. Variables vs. objects (a) declaration and initialization of an `int` variable (b) instantiation of an `Integer` object

Clearly `x` and `y` differ by more than their values: `x` is a variable that holds a value; `y` is an object variable that holds a reference to an object. As noted earlier, data fields in objects are not, in general, directly accessible. So, the following statement using `x` and `y` as declared above is not allowed:

```
int z = x + y;  // wrong!
```

The data field in an `Integer` object is only accessible using the methods of the `Integer` class. One such method — the `intValue()` method — returns an `int` equal to the value of the object, effectively "unwrapping" the `Integer` object:

```
int z = x + y.intValue();  // OK!
```

Note the format of the method call. The `intValue()` method is an instance method, because it is called through an instance of the class — an `Integer` object. The syntax uses dot notation, where the name of the object variable precedes the dot.

Some of Java's classes include only instance methods, some include only class methods, some include both. The `Integer` class includes both instance methods and class methods. The class methods provide useful services; however, they are not called through instances of the class. But, this is not new. We met a class method of the `Integer` class earlier. The statement

```
int x = Integer.parseInt("1234");
```

converts the string "`1234`" into the integer 1,234 and assigns the result to the `int` variable `x`. The method `parseInt()` is a class method. It is *not* called through an instance of the class.

Although dot notation is used above, the term "`Integer.`" identifies the class where the method is defined, rather than name of an `Integer` object. This is one of the trickier aspects of distinguishing instance methods from class methods. For an instance method, the dot is preceded by the name of an object variable. For a class method, the dot is preceded by the name of a class.

It is interesting that the `parseInt()` method accepts a `String` argument and returns an `int`. Neither the argument nor the returned value is an `Integer` object. So, the earlier comment that class methods "provide useful services" is quite true. They do not operate on instance variables; they provide services that are of general use. The `parseInt()` method converts a `String` to an `int`, and this is a useful service — one we have called upon in previous programs.

Another useful `Integer` class method is the `toString()` method. The following statement

```
String s = Integer.toString(1234);
```

converts the `int` constant 1,234 to a `String` object and returns a reference to the object. We have already met this method through the following shorthand notation:

```
String s = "" + 1234;
```

In general, the data fields in objects are private and are only accessible through methods of the class. However, class definitions sometimes include constants that are "public", and, therefore, accessible anywhere the class is accessible. Java's wrapper classes include publicly defined constants identifying the range for each type. For example, the `Integer` class has constants called `MIN_VALUE` and `MAX_VALUE` equal to 0x80000000 and 0x7fffffff respectively. These values are the hexadecimal equivalent of the most-negative and most-positive values represented by 32-bit integers. The table of ranges shown earlier was obtained from a simple program called `FindRanges` that prints these constants. The listing is given here in Figure 2.

```
public class FindRanges
{
   public static void main(String[] args)
   {
      System.out.println("Integer range:");
      System.out.println("   min: " + Integer.MIN_VALUE);
      System.out.println("   max: " + Integer.MAX_VALUE);
      System.out.println("Double range:");
      System.out.println("   min: " + Double.MIN_VALUE);
      System.out.println("   max: " + Double.MAX_VALUE);
      System.out.println("Long range:");
      System.out.println("   min: " + Long.MIN_VALUE);
      System.out.println("   max: " + Long.MAX_VALUE);
      System.out.println("Short range:");
      System.out.println("   min: " + Short.MIN_VALUE);
      System.out.println("   max: " + Short.MAX_VALUE);
      System.out.println("Byte range:");
      System.out.println("   min: " + Byte.MIN_VALUE);
      System.out.println("   max: " + Byte.MAX_VALUE);
      System.out.println("Float range:");
      System.out.println("   min: " + Float.MIN_VALUE);
      System.out.println("   max: " + Float.MAX_VALUE);
   }
}
```

Figure 2. `FindRanges.java`

This program generates the following output:

```
Integer range:
   min: -2147483648
   max: 2147483647
Double range:
   min: 4.9E-324
   max: 1.7976931348623157E308
Long range:
   min: -9223372036854775808
   max: 9223372036854775807
Short range:
   min: -32768
   max: 32767
Byte range:
   min: -128
   max: 127
Float range:
   min: 1.4E-45
   max: 3.4028235E38
```

In the case of `float` and `double,` the minimum and maximum values are those closest to plus or minus zero or plus or minus infinity, respectively.

The most common methods of the `Integer` wrapper class are summarized in Table 1. Similar methods for the other wrapper classes are found in the Java API documentation.

Table 1. Methods of the Integer Wrapper Class

Method	Purpose
Constructors	
`Integer(i)`	constructs an `Integer` object equivalent to the integer `i`
`Integer(s)`	constructs an `Integer` object equivalent to the string `s`
Class Methods	
`parseInt(s)`	returns a signed decimal integer value equivalent to string `s`
`toString(i)`	returns a new `String` object representing the integer `i`
Instance Methods	
`byteValue()`	returns the value of this `Integer` as a `byte`
`doubleValue()`	returns the value of this `Integer` as an `double`
`floatValue()`	returns the value of this `Integer` as a `float`
`intValue()`	returns the value of this `Integer` as an `int`
`longValue()`	returns the value of this `Integer` as a `long`
`shortValue()`	returns the value of this `Integer` as a `short`
`toString()`	returns a `String` object representing the value of this `Integer`

In the descriptions of the instance methods, the phrase "this `Integer`" refers to the instance variable on which the method is acting. For example, if `y` is an `Integer` object variable, the expression `y.doubleValue()` returns the value of "this `Integer`", or `y`, as a `double`.