```
// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

For example,

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

`Integer.parseInt("12", 2)` would raise a runtime exception because `12` is not a binary number.

14.10 Sorting an Array of Objects

This example presents a static generic method for sorting an array of comparable objects. The objects are instances of the `Comparable` interface, and they are compared using the `compareTo` method. The method can be used to sort an array of any objects as long as their classes implement the `Comparable` interface.

To test the method, the program sorts an array of integers, an array of double numbers, an array of characters, and an array of strings. The program is shown in Listing 14.10.

LISTING 14.10 GenericSort.java

```
1 public class GenericSort {
2   public static void main(String[] args) {
3     // Create an Integer array
4     Integer[] intArray = {new Integer(2), new Integer(4),
5       new Integer(3)};
6
7     // Create a Double array
8     Double[] doubleArray = {new Double(3.4), new Double(1.3),
9       new Double(-22.1)};
10
11    // Create a Character array
12    Character[] charArray = {new Character('a'),
13      new Character('J'), new Character('r')};
14
15    // Create a String array
16    String[] stringArray = {"Tom", "John", "Fred"};
17
18    // Sort the arrays
19    sort(intArray);                                    sort Integer objects
20    sort(doubleArray);                                 sort Double objects
21    sort(charArray);                                   sort Character objects
22    sort(stringArray);                                 sort String objects
23
24    // Display the sorted arrays
25    System.out.print("Sorted Integer objects: ");
26    printList(intArray);
27    System.out.print("Sorted Double objects: ");
28    printList(doubleArray);
29    System.out.print("Sorted Character objects: ");
30    printList(charArray);
31    System.out.print("Sorted String objects: ");
32    printList(stringArray);
33  }
34
```

generic sort method

compareTo

```
35   /** Sort an array of comparable objects */
36   public static void sort(Comparable[] list) {
37     Comparable currentMin;
38     int currentMinIndex;
39
40     for (int i = 0; i < list.length - 1; i++) {
41       // Find the maximum in the list[0..i]
42       currentMin = list[i];
43       currentMinIndex = i;
44
45       for (int j = i + 1; j < list.length; j++) {
46         if (currentMin.compareTo(list[j]) > 0) {
47           currentMin = list[j];
48           currentMinIndex = j;
49         }
50       }
51
52       // Swap list[i] with list[currentMinIndex] if necessary;
53       if (currentMinIndex != i) {
54         list[currentMinIndex] = list[i];
55         list[i] = currentMin;
56       }
57     }
58   }
59
60   /** Print an array of objects */
61   public static void printList(Object[] list) {
62     for (int i = 0; i < list.length; i++)
63       System.out.print(list[i] + " ");
64     System.out.println();
65   }
66 }
```

```
Sorted Integer objects: 2 3 4
Sorted Double objects: -22.1 1.3 3.4
Sorted Character objects: J a r
Sorted String objects: Fred John Tom
```

The algorithm for the **sort** method is the same as in §6.10.1, "Selection Sort." The **sort** method in §6.10.1 sorts an array of **double** values. The **sort** method in this example can sort an array of any object type, provided that the objects are also instances of the **Comparable** interface. This is another example of *generic programming*. Generic programming enables a method to operate on arguments of generic types, making it reusable with multiple types.

Integer, **Double**, **Character**, and **String** implement **Comparable**, so the objects of these classes can be compared using the **compareTo** method. The **sort** method uses the **compareTo** method to determine the order of the objects in the array.

Tip

Arrays.sort method

Java provides a static **sort** method for sorting an array of any object type in the **java.util.Arrays** class, provided that the elements in the array are comparable. Thus you can use the following code to sort arrays in this example:

```
java.util.Arrays.sort(intArray);
java.util.Arrays.sort(doubleArray);
java.util.Arrays.sort(charArray);
java.util.Arrays.sort(stringArray);
```

Note

Arrays are objects. An array is an instance of the `Object` class. Furthermore, if A is a subtype of B, every instance of `A[]` is an instance of `B[]`. Therefore, the following statements are all true:

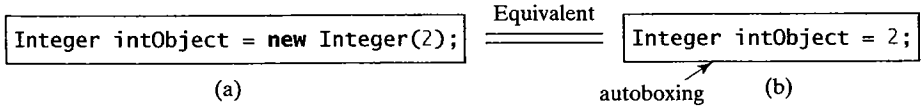
```
new int[10] instanceof Object
new Integer[10] instanceof Object
new Integer[10] instanceof Comparable[]
new Integer[10] instanceof Number[]
new Number[10] instanceof Object[]
```

Caution

Although an `int` value can be assigned to a `double` type variable, `int[]` and `double[]` are two incompatible types. Therefore, you cannot assign an `int[]` array to a variable of `double[]` or `Object[]` type.

14.11 Automatic Conversion between Primitive Types and Wrapper Class Types

Java allows primitive types and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b) due to autoboxing:



Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*. The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value. Consider the following example:

boxing
unboxing

```
1 Integer[] intArray = {1, 2, 3};
2 System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

In line 1, primitive values 1, 2, and 3 are automatically boxed into objects `new Integer(1)`, `new Integer(2)`, and `new Integer(3)`. In line 2, objects `intArray[0]`, `intArray[1]`, and `intArray[2]` are automatically converted into `int` values that are added together.

14.12 The `BigInteger` and `BigDecimal` Classes

If you need to compute with very large integers or high-precision floating-point values, you can use the `BigInteger` and `BigDecimal` classes in the `java.math` package. Both are *immutable*. Both extend the `Number` class and implement the `Comparable` interface. The largest integer of the `long` type is `Long.MAX_VALUE` (i.e., `9223372036854775807`). An instance of `BigInteger` can represent an integer of any size. You can use `new BigInteger(String)` and `new BigDecimal(String)` to create an instance of `BigInteger` and `BigDecimal`, use the `add`, `subtract`, `multiple`, `divide`, and `remainder` methods to perform arithmetic operations, and use the `compareTo` method to compare two big numbers. For example, the following code creates two `BigInteger` objects and multiplies them.

immutable

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```