20.7 Wildcards in Methods That Accept Type Parameters

In this section, we introduce a powerful generics concept known as **wildcards**. Let's consider an example that motivates wildcards (Fig. 20.10). Suppose that you'd like to implement a generic method `sum` that totals the numbers in a collection, such as a `List`. You'd begin by inserting the numbers in the collection. Because generic classes can be used only with class or interface types, the numbers would be *autoboxed* as objects of the type-wrapper classes. For example, any `int` value would be *autoboxed* as an `Integer` object, and any `double` value would be *autoboxed* as a `Double` object. We'd like to be able to total all the numbers in the `List` regardless of their type. For this reason, we'll declare the `List` with the type argument `Number`, which is the superclass of both `Integer` and `Double`. In addition, method `sum` will receive a parameter of type `List<Number>` and total its elements.

```
1  // Fig. 20.10: TotalNumbers.java
2  // Totaling the numbers in a List<Number>.
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class TotalNumbers {
7     public static void main(String[] args) {
```

```
 8          // create, initialize and output List of N
 9          // both Integers and Doubles, then display
10          Number[] numbers = {1, 2.4, 3, 4.1}; // In
11          List<Number> numberList = new ArrayList<>(
                       12
   13          for (Number element : numbers) {
14              numberList.add(element); // place each
                    15          }
                          16
17          System.out.printf("numberList contains: %s
18          System.out.printf("Total of the elements i
          19              sum(numberList));
                      20      }
                          21
   22       // calculate total of List elements
23       public static double sum(List<Number> list) {
24          double total = 0; // initialize total
                          25
            26          // calculate sum
     27           for (Number element : list) {
  28              total += element.doubleValue();
                  29          }
                          30
              31          return total;
                      32      }
                        33   }
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 20.10

Totaling the numbers in a List<Number>.

Line 10 declares and initializes an array of `Number`s. Because the initializers are primitive values, Java *autoboxes* each primitive value as an object of its corresponding wrapper type. The `int` values `1` and `3` are *autoboxed* as `Integer` objects, and the `double` values `2.4` and `4.1` are *autoboxed* as `Double` objects. Line 11 creates an `ArrayList` object that stores `Number`s and assigns it to `List` variable `numberList`.

Lines 13–15 traverse array `numbers` and place each element in `numberList`. Line 17 outputs the `List`'s contents by implicitly invoking the `List`'s `toString` method. Lines 18–19 display the sum of the elements that is returned by the call to method `sum`.

Method `sum` (lines 23–32) receives a `List` of `Number`s and calculates the total of its `Number`s. The method uses `double` values to perform the calculations and returns the result as a `double`. Lines 27–29 total the `List`'s elements. The `for` statement assigns each `Number` to variable `element`, then uses `Number` **method** `doubleValue` to obtain the `Number`'s underlying primitive value as a `double`. The result is added to `total`. When the loop terminates, line 31 returns the `total`.

Implementing Method `sum` with a Wildcard Type

Argument in Its Parameter

Recall that the purpose of method `sum` in Fig. 20.10 was to total any type of `Number`s stored in a `List`. We created a `List` of `Number`s that contained both `Integer` and `Double` objects. The output of Fig. 20.10 demonstrates that method `sum` worked properly. Given that method `sum` can total the elements of a `List` of `Number`s, you might expect that the method would also work for `List`s that contain elements of only one numeric type, such as `List<Integer>`. So we modified class `TotalNumbers` to create a `List` of `Integer`s and pass it to method `sum`. When we compile the program, the compiler issues the following error message:

```
TotalNumbersErrors.java:19: error: incompatible types
List<Integer> cannot be converted to List<Number>
```

Although `Number` is the superclass of `Integer`, the compiler doesn't consider the type `List<Number>` to be a supertype of `List<Integer>`. If it were, then every operation we could perform on a `List<Number>` would also work on a `List<Integer>`. Consider the fact that you can add a `Double` object to a `List<Number>` because a `Double` *is a* `Number`, but you cannot add a `Double` object to a `List<Integer>` because a `Double` *is not an* `Integer`. Thus, the subtype relationship does not hold.

How do we create a more flexible version of method `sum` that

can total the elements of any `List` containing elements of any subclass of `Number`? This is where **wildcard type arguments** are important. Wildcards enable you to specify method parameters, return values, variables or fields, and so on, that act as supertypes or subtypes of parameterized types. In Fig. 20.11, method `sum`'s parameter is declared in line 52 with the type:

```
List<? extends Number>
```

A wildcard type argument is denoted by a question mark (`?`), which represents an "unknown type." In this case, the wildcard `extends` class `Number`, which means that the wildcard has an upper bound of `Number`. Thus, the unknown-type argument must be either `Number` or a subclass of `Number`. With the wildcard type argument, method `sum` can receive an argument a `List` containing any type of `Number`, such as a `List<Integer>` (line 20), `List<Double>` (line 34) or `List<Number>` (line 48).

```
 1  // Fig. 20.11: WildcardTest.java
 2  // Wildcard test program.
 3  import java.util.ArrayList;
 4  import java.util.List;
 5
 6  public class WildcardTest {
 7     public static void main(String[] args) {
 8        // create, initialize and output List of I
 9        // display total of the elements
10        Integer[] integers = {1, 2, 3, 4, 5};
11        List<Integer> integerList = new ArrayList<
```

```
                    12
  13         // insert elements in integerList
  14         for (Integer element : integers) {
     15            integerList.add(element);
                   16         }
                        17
18         System.out.printf("integerList contains: %
19         System.out.printf("Total of the elements i
        20            sum(integerList));
                        21
22         // create, initialize and output List of D
   23         // display total of the elements
 24         Double[] doubles = {1.1, 3.3, 5.5};
25         List<Double> doubleList = new ArrayList<>(
                        26
   27         // insert elements in doubleList
   28         for (Double element : doubles) {
      29            doubleList.add(element);
                   30         }
                        31
32         System.out.printf("doubleList contains: %s
33         System.out.printf("Total of the elements i
        34            sum(doubleList));
                        35
36         // create, initialize and output List of N
37         // both Integers and Doubles, then display
38         Number[] numbers = {1, 2.4, 3, 4.1}; // In
39         List<Number> numberList = new ArrayList<>(
                        40
   41         // insert elements in numberList
   42         for (Number element : numbers) {
      43            numberList.add(element);
                   44         }
                        45
46         System.out.printf("numberList contains: %s
47         System.out.printf("Total of the elements i
        48            sum(numberList));
                    49      }
                         50
51      // total the elements; using a wildcard in th
```

```
52      public static double sum(List<? extends Numbe
53         double total = 0; // initialize total
                           54
              55          // calculate sum
     56          for (Number element : list) {
  57              total += element.doubleValue();
                    58          }
                           59
               60          return total;
                      61       }
                        62   }
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 20.11

Wildcard test program.

Lines 10–20 create and initialize a `List<Integer>`, output its elements and total them by calling method `sum` (line 20). Lines 24–34 and 38–48 perform the same operations for a `List<Double>` and a `List<Number>` that contains `Integer`s and `Double`s.

In method `sum` (lines 52–61), although the `List` argument's element types are not directly known by the method, they're known to be at least of type `Number`, because the wildcard was specified with the upper bound `Number`. For this reason line 57 is allowed, because all `Number` objects have a `doubleValue` method.

Wildcard Restrictions

Because the wildcard (`?`) in the method's header (line 52) does not specify a type-parameter name, you cannot use it as a type name throughout the method's body (i.e., you cannot replace `Number` with `?` in line 56). You could, however, declare method `sum` as follows:

```
   public static <T extends Number> double sum(List<T> l
```

which allows the method to receive a `List` that contains elements of any `Number` subclass. You could then use the type parameter `T` throughout the method body.

If the wildcard is specified without an upper bound, then only the methods of type `Object` can be invoked on values of the wildcard type. Also, methods that use wildcards in their parameter's type arguments cannot be used to add elements to a collection referenced by the parameter.

Common Programming Error 20.3

Using a wildcard in a method's type-parameter section or using a wildcard as an explicit type of a variable in the method body is a syntax error.