20.3 Generic Methods: Implementation and Compile-Time Translation

If the operations performed by several overloaded methods are *identical* for each argument type, the overloaded methods can be more conveniently coded using a generic method. You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. At *compilation time*, the compiler ensures the *type safety* of your code, preventing many runtime errors.

Figure 20.3 reimplements Fig. 20.1 using a generic `printArray` method (lines 20–27 of Fig. 20.3). The `printArray` calls in lines 12, 14 and 16 are identical to those of Fig. 20.1, and the outputs of the two applications are identical. This demonstrates the expressive power of generics.

```
1   // Fig. 20.3: GenericMethodTest.java
2   // Printing array elements using generic method
3
4   public class GenericMethodTest {
5     public static void main(String[] args) {
6       // create arrays of Integer, Double and Ch
7       Integer[] integerArray = {1, 2, 3, 4, 5};
```

```
 8          Double[] doubleArray = {1.1, 2.2, 3.3, 4.4
 9          Character[] characterArray = {'H', 'E', 'L
                       10
11          System.out.printf("Array integerArray cont
12          printArray(integerArray); // pass an Integ
13          System.out.printf("Array doubleArray conta
14          printArray(doubleArray); // pass a Double
15          System.out.printf("Array characterArray co
16          printArray(characterArray); // pass a Char
                      17      }
                       18
       19      // generic method printArray
20       public static <T> void printArray(T[] inputAr
       21         // display array elements
    22          for (T element : inputArray) {
 23            System.out.printf("%s ", element);
                     24         }
                       25
         26          System.out.println();
                      27      }
                        28   }
```

```
     Array integerArray contains: 1 2 3 4 5
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7
     Array characterArray contains: H E L L O
```

Fig. 20.3

Printing array elements using generic method printArray.

Type Parameter Section of a Generic Method

All generic method declarations have a **type-parameter section** (line 20; `<T>` in this example) delimited by **angle brackets** that precedes the method's return type. Each type-parameter section contains one or more **type parameters**, separated by commas. A type parameter, also known as a **type variable**, is an identifier that specifies a generic type name. The type parameters can be used to declare the return type, parameters and local variables in a generic method declaration, and they act as placeholders for the types of the arguments passed to the generic method, which are known as **actual type arguments**. A generic method's body is declared like that of any other method. *Type parameters can represent only reference types*—not primitive types (like `int`, `double` and `char`). Also, the type-parameter names throughout the method declaration must match those declared in the type-parameter section. For example, line 22 declares `element` as type `T`, which matches the type parameter (`T`) declared in line 20. A type parameter can be declared only once in the type-parameter section but can appear more than once in the method's parameter list. For example, the type-parameter name `T` appears twice in the following method's parameter list:

```
public static <T> T maximum(T value1, T value2)
```

Type-parameter names need not be unique among different generic methods. In method `printArray`, `T` appears in the same two locations where the overloaded `printArray` methods of Fig. 20.1 specified `Integer`, `Double` or `Character` as the array element type. The remainder of `printArray` is identical to the versions presented in Fig. 20.1.

Good Programming Practice 20.1

The letters `T` *(for "type"),* `E` *(for "element"),* `K` *(for "key") and* `V` *(for "value") are commonly used as type parameters. For other common ones, see* `http://docs.oracle.com/javase/tutorial/java/generics/types.html`.

Testing the Generic `printArray` Method

As in Fig. 20.1, the program in Fig. 20.3 begins by declaring and initializing six-element `Integer` array `integerArray`

(line 7), seven-element `Double` array `doubleArray` (line 8) and five-element `Character` array `characterArray` (line 9). Then each array is output by calling `printArray` (lines 12, 14 and 16)—once with argument `integerArray`, once with argument `doubleArray` and once with argument `characterArray`.

When the compiler encounters line 12, it first determines argument `integerArray`'s type (i.e., `Integer[]`) and attempts to locate a method named `printArray` that specifies a single `Integer[]` parameter. There's no such method in this example. Next, the compiler determines whether there's a generic method named `printArray` that specifies a single array parameter and uses a type parameter to represent the array element type. The compiler determines that `printArray` (lines 20–27) is a match and sets up a call to the method. The same process is repeated for the calls to method `printArray` at lines 14 and 16.

Common Programming Error 20.1

If the compiler cannot match a method call to a nongeneric or a generic method declaration, a compilation error occurs.

Common Programming Error 20.2

If the compiler doesn't find a method declaration that matches a method call exactly, but does find two or more methods that can satisfy the method call, a compilation error occurs. For the complete details of resolving calls to overloaded and generic methods, see `http://docs.oracle.com/javase/specs/jls/se8/html/jls15.html#jls-15.12`.

In addition to setting up the method calls, the compiler also determines whether the operations in the method body can be applied to elements of the type stored in the array argument. The only operation performed on the array elements in this example is to output their `String` representation. Line 23 performs an *implicit* `toString` *call* on every `element`. *To work with generics, every element of the array must be an object of a class or interface type.* Since all objects have a `toString` method, the compiler is satisfied that line 23 performs a *valid* operation for any object in `printArray`'s array argument. The `toString` methods of classes `Integer`, `Double` and `Character` return the `String` representations of the underlying `int`, `double` or `char` value, respectively.

Erasure at Compilation

Time

When the compiler translates generic method `printArray` into Java bytecodes, it removes the type-parameter section and *replaces the type parameters with actual types*. This process is known as **erasure**. By default all generic types are replaced with type `Object`. So the compiled version of method `printArray` appears as shown in Fig. 20.4—there's only *one* copy of this code, which is used for all `printArray` calls in the example. This is quite different from similar mechanisms in other programming languages, such as C++'s templates, in which a *separate copy of the source code* is generated and compiled for *every* type passed as an argument to the method. As you'll see in Section 20.4, the translation and compilation of generics is a bit more involved than what we've discussed in this section.

By declaring `printArray` as a generic method in Fig. 20.3, we eliminated the need for the overloaded methods of Fig. 20.1 and created a reusable method that can output the `String` representations of the elements in any array that contains objects. However, this particular example could have simply declared the `printArray` method as shown in Fig. 20.4, using an `Object` array as the parameter. This would have yielded the same results, because any `Object` can be output as a `String`. In a generic method, the benefits become more apparent when you place restrictions on the type parameters, as we demonstrate in the next section.

```
1   public static void printArray(Object[] inputArra
```

```
        2      // display array elements
 3      for (Object element : inputArray) {
4         System.out.printf("%s ", element);
                  5      }
                      6
         7      System.out.println();
                     8   }
```

Fig. 20.4

Generic method printArray after the compiler performs erasure.