

20.2 Motivation for Generic Methods

Overloaded methods are often used to perform *similar* operations on *different* types of data. To motivate generic methods, let's begin with an example (Fig. 20.1) containing overloaded `printArray` methods (lines 20–27, 30–37 and 40–47) that print the `String` representations of the elements of an `Integer` array, a `Double` array and a `Character` array, respectively. We could have used arrays of primitive types `int`, `double` and `char`. We're using arrays of the type-wrapper classes to set up our generic method example, because *only reference types can be used to specify generic types in generic methods and classes*.

```
1 // Fig. 20.1: OverloadedMethods.java
2 // Printing array elements using overloaded method
3
4 public class OverloadedMethods {
5     public static void main(String[] args) {
6         // create arrays of Integer, Double and Ch
7         Integer[] integerArray = {1, 2, 3, 4, 5, 6
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     // method printArray to print Integer array
20     public static void printArray(Integer[] input
21         // display array elements
22         for (Integer element : inputArray) {
23             System.out.printf("%s ", element);
24         }
25
26         System.out.println();
27     }
28
29     // method printArray to print Double array
30     public static void printArray(Double[] inputA
31         // display array elements
32         for (Double element : inputArray) {
33             System.out.printf("%s ", element);
34         }
35
36         System.out.println();
37     }
38
39     // method printArray to print Character array
40     public static void printArray(Character[] inp
41         // display array elements
42         for (Character element : inputArray) {
43             System.out.printf("%s ", element);
44         }
45
46         System.out.println();
47     }
48 }
```



```
Array integerArray contains: 1 2 3 4 5 6
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. 20.1

Printing array elements using overloaded methods.

The program begins by declaring and initializing three arrays —six-element **Integer** array **integerArray** (line 7), seven-element **Double** array **doubleArray** (line 8) and five-element **Character** array **characterArray** (line 9). Then lines 11–16 display the contents of each array.

When the compiler encounters a method call, it attempts to locate a method declaration with the same name and with parameters that match the argument types in the call. In this example, each **printArray** call matches one of the **printArray** method declarations. For example, line 12 calls **printArray** with **integerArray** as its argument. The compiler determines the argument's type (i.e., **Integer**[]) and attempts to locate a **print-Array** method that specifies an **Integer**[] parameter (lines 20–27), then sets up a call to that method. Similarly, when the compiler encounters the call at line 14, it determines the argument's type (i.e., **Double**[]), then attempts to locate a **printArray** method that specifies a **Double**[] parameter (lines 30–37), then sets up a call to that method. Finally, when the compiler encounters the call at line 16, it determines the argument's type (i.e., **Character**[]), then attempts to locate a **printArray** method that specifies a **Character**[] parameter (lines 40–47), then sets up a call to that method.

Common Features in the Overloaded `printArray` Methods

Study each `printArray` method. The array element type appears in each method's header (lines 20, 30 and 40) and `for`-statement header (lines 22, 32 and 42). If we were to replace the element types in each method with a generic name —`T` by convention—then all three methods would look like the one in [Fig. 20.2](#). It appears that if we can replace the array element type in each of the three methods with a *single generic type*, then we should be able to declare *one* `printArray` method that can display the `String` representations of the elements of *any* array that contains objects. The method in [Fig. 20.2](#) is similar to the generic `printArray` method declaration you'll see in [Section 20.3](#). The one shown here *will not compile*—we use this simply to show that the three `printArray` methods of [Fig. 20.1](#) are identical except for the types they process.

```
1  public static void printArray(T [] inputArray) {
2      // display array elements
3      for (T element : inputArray) {
4          System.out.printf("%s ", element);
5      }
6
7      System.out.println();
8  }
```

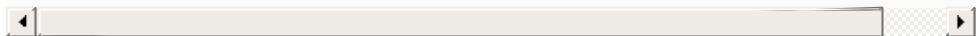


Fig. 20.2

printArray method in which actual type names are replaced with a generic type name (in this case T).