

Chapter 20

Generic Classes and Methods: A Deeper Look

Java How to Program, 11/e, Global Edition
Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Create generic methods that perform identical tasks on arguments of different types.
- Create a generic **Stack** class that can be used to store objects of any class or interface type.
- Learn about compile-time translation of generic methods and classes.
- Learn how to overload generic methods with non-generic or generic methods.
- Use wildcards when precise type information about a parameter is not required in the method body.

-
- 20.1** Introduction
 - 20.2** Motivation for Generic Methods
 - 20.3** Generic Methods: Implementation and Compile-Time Translation
 - 20.4** Additional Compile-Time Translation Issues:
Methods That Use a Type Parameter as the Return Type
 - 20.5** Overloading Generic Methods
 - 20.6** Generic Classes
 - 20.7** Wildcards in Methods That Accept Type Parameters
 - 20.8** Wrap-Up
-

```
1 // Fig. 20.1: OverloadedMethods.java
2 // Printing array elements using overloaded methods.
3
4 public class OverloadedMethods {
5     public static void main(String[] args) {
6         // create arrays of Integer, Double and Character
7         Integer[] integerArray = {1, 2, 3, 4, 5, 6};
8         Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
9         Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};
10
11     System.out.printf("Array integerArray contains: ");
12     printArray(integerArray); // pass an Integer array
13     System.out.printf("Array doubleArray contains: ");
14     printArray(doubleArray); // pass a Double array
15     System.out.printf("Array characterArray contains: ");
16     printArray(characterArray); // pass a Character array
17 }
```

Fig. 20.1 | Printing array elements using overloaded methods. (Part I of 3.)

```
18
19 // method printArray to print Integer array
20 public static void printArray(Integer[] inputArray) {
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[] inputArray) {
31     // display array elements
32     for (Double element : inputArray) {
33         System.out.printf("%s ", element);
34     }
35
36     System.out.println();
37 }
```

Fig. 20.1 | Printing array elements using overloaded methods. (Part 2 of 3.)

```
38
39 // method printArray to print Character array
40 public static void printArray(Character[] inputArray) {
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 7.7
Array characterArray contains: H E L L O
```

Fig. 20.1 | Printing array elements using overloaded methods. (Part 3 of 3.)

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

Fig. 20.2 | printArray method in which actual type names are replaced with a generic type name (in this case T).

```
1 // Fig. 20.3: GenericMethodTest.java
2 // Printing array elements using generic method printArray.
3
4 public class GenericMethodTest {
5     public static void main(String[] args) {
6         // create arrays of Integer, Double and Character
7         Integer[] integerArray = {1, 2, 3, 4, 5};
8         Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
9         Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};
10
11     System.out.printf("Array integerArray contains: ");
12     printArray(integerArray); // pass an Integer array
13     System.out.printf("Array doubleArray contains: ");
14     printArray(doubleArray); // pass a Double array
15     System.out.printf("Array characterArray contains: ");
16     printArray(characterArray); // pass a Character array
17 }
```

Fig. 20.3 | Printing array elements using generic method `printArray`. (Part I of 2.)

```
18  
19      // generic method printArray  
20      public static <T> void printArray(T[] inputArray) {  
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 7.7  
Array characterArray contains: H E L L O
```

Fig. 20.3 | Printing array elements using generic method `printArray`. (Part 2 of 2.)



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>.



Common Programming Error 20.1

If the compiler cannot match a method call to a non-generic 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/jls-15.html#jls-15.12>.

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

Fig. 20.4 | Generic method `printArray` after the compiler performs erasure.

```
1 // Fig. 20.5: MaximumTest.java
2 // Generic method maximum returns the largest of three objects.
3
4 public class MaximumTest {
5     public static void main(String[] args) {
6         System.out.printf("Maximum of %d, %d and %d is %d%n", 3, 4, 5,
7                           maximum(3, 4, 5));
8         System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f%n",
9                           6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
10        System.out.printf("Maximum of %s, %s and %s is %s%n", "pear",
11                          "apple", "orange", maximum("pear", "apple", "orange"));
12    }
13}
```

Fig. 20.5 | Generic method `maximum` with an upper bound on its type parameter. (Part I of 2.)

```
14 // determines the largest of three Comparable objects
15 public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
16     T max = x; // assume x is initially the largest
17
18     if (y.compareTo(max) > 0) {
19         max = y; // y is the largest so far
20     }
21
22     if (z.compareTo(max) > 0) {
23         max = z; // z is the largest
24     }
25
26     return max; // returns the largest object
27 }
28 }
```

Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear

Fig. 20.5 | Generic method `maximum` with an upper bound on its type parameter. (Part 2 of 2.)

```
1 public static Comparable maximum(Comparable x, Comparable y,
2     Comparable z) {
3
4     Comparable max = x; // assume x is initially the largest
5
6     if (y.compareTo(max) > 0) {
7         max = y; // y is the largest so far
8     }
9
10    if (z.compareTo(max) > 0) {
11        max = z; // z is the largest
12    }
13
14    return max; // returns the largest object
15 }
```

Fig. 20.6 | Generic method `maximum` after erasure is performed by the compiler.

```
1 // Fig. 20.7: Stack.java
2 // Stack generic class declaration.
3 import java.util.ArrayList;
4 import java.util.NoSuchElementException;
5
6 public class Stack<E> {
7     private final ArrayList<E> elements; // ArrayList stores stack elements
8
9     // no-argument constructor creates a stack of the default size
10    public Stack() {
11        this(10); // default stack size
12    }
13
14    // constructor creates a stack of the specified number of elements
15    public Stack(int capacity) {
16        int initCapacity = capacity > 0 ? capacity : 10; // validate
17        elements = new ArrayList<E>(initCapacity); // create ArrayList
18    }
}
```

Fig. 20.7 | Stack generic class declaration. (Part I of 2.)

```
19
20     // push element onto stack
21     public void push(E pushValue) {
22         elements.add(pushValue); // place pushValue on Stack
23     }
24
25     // return the top element if not empty; else throw exception
26     public E pop() {
27         if (elements.isEmpty()) { // if stack is empty
28             throw new NoSuchElementException("Stack is empty, cannot pop");
29         }
30
31         // remove and return top element of Stack
32         return elements.remove(elements.size() - 1);
33     }
34 }
```

Fig. 20.7 | Stack generic class declaration. (Part 2 of 2.)

```
1 // Fig. 20.8: StackTest.java
2 // Stack generic class test program.
3 import java.util.NoSuchElementException;
4
5 public class StackTest {
6     public static void main(String[] args) {
7         double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8         int[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10        // Create a Stack<Double> and a Stack<Integer>
11        Stack<Double> doubleStack = new Stack<>(5);
12        Stack<Integer> integerStack = new Stack<>();
13
14        // push elements of doubleElements onto doubleStack
15        testPushDouble(doubleStack, doubleElements);
16        testPopDouble(doubleStack); // pop from doubleStack
17
18        // push elements of integerElements onto integerStack
19        testPushInteger(integerStack, integerElements);
20        testPopInteger(integerStack); // pop from integerStack
21    }
```

Fig. 20.8 | Stack generic class test program. (Part I of 6.)

```
22
23     // test push method with double stack
24     private static void testPushDouble(
25         Stack<Double> stack, double[] values) {
26         System.out.printf("%nPushing elements onto doubleStack%n");
27
28         // push elements to Stack
29         for (double value : values) {
30             System.out.printf("%.1f ", value);
31             stack.push(value); // push onto doubleStack
32         }
33     }
34 }
```

Fig. 20.8 | Stack generic class test program. (Part 2 of 6.)

```
35     // test pop method with double stack
36     private static void testPopDouble(Stack<Double> stack) {
37         // pop elements from stack
38         try {
39             System.out.printf("%nPopping elements from doubleStack%n");
40             double popValue; // store element removed from stack
41
42             // remove all elements from Stack
43             while (true) {
44                 popValue = stack.pop(); // pop from doubleStack
45                 System.out.printf("%.1f ", popValue);
46             }
47         }
48         catch(NoSuchElementException noSuchElementException) {
49             System.err.println();
50             noSuchElementException.printStackTrace();
51         }
52     }
```

Fig. 20.8 | Stack generic class test program. (Part 3 of 6.)

```
53
54     // test push method with integer stack
55     private static void testPushInteger(
56         Stack<Integer> stack, int[] values) {
57         System.out.printf("%nPushing elements onto integerStack%n");
58
59         // push elements to Stack
60         for (int value : values) {
61             System.out.printf("%d ", value);
62             stack.push(value); // push onto integerStack
63         }
64     }
65
```

Fig. 20.8 | Stack generic class test program. (Part 4 of 6.)

```
66     // test pop method with integer stack
67     private static void testPopInteger(Stack<Integer> stack) {
68         // pop elements from stack
69         try {
70             System.out.printf("%nPopping elements from integerStack%n");
71             int popValue; // store element removed from stack
72
73             // remove all elements from Stack
74             while (true) {
75                 popValue = stack.pop(); // pop from intStack
76                 System.out.printf("%d ", popValue);
77             }
78         }
79         catch(NoSuchElementException noSuchElementException) {
80             System.err.println();
81             noSuchElementException.printStackTrace();
82         }
83     }
84 }
```

Fig. 20.8 | Stack generic class test program. (Part 5 of 6.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest.testPopDouble(StackTest.java:44)
    at StackTest.main(StackTest.java:16)
```

```
Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest.testPopInteger(StackTest.java:75)
    at StackTest.main(StackTest.java:20)
```

Fig. 20.8 | Stack generic class test program. (Part 6 of 6.)

```
1 // Fig. 20.9: StackTest2.java
2 // Passing generic Stack objects to generic methods.
3 import java.util.NoSuchElementException;
4
5 public class StackTest2 {
6     public static void main(String[] args) {
7         Double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8         Integer[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10        // Create a Stack<Double> and a Stack<Integer>
11        Stack<Double> doubleStack = new Stack<>(5);
12        Stack<Integer> integerStack = new Stack<>();
13
14        // push elements of doubleElements onto doubleStack
15        testPush("doubleStack", doubleStack, doubleElements);
16        testPop("doubleStack", doubleStack); // pop from doubleStack
17
18        // push elements of integerElements onto integerStack
19        testPush("integerStack", integerStack, integerElements);
20        testPop("integerStack", integerStack); // pop from integerStack
21    }
```

Fig. 20.9 | Passing generic Stack objects to generic methods. (Part I of 4.)

```
22
23 // generic method testPush pushes elements onto a Stack
24 public static <E> void testPush(String name , Stack<E> stack,
25     E[] elements) {
26     System.out.printf("%nPushing elements onto %s%n", name);
27
28     // push elements onto Stack
29     for (E element : elements) {
30         System.out.printf("%s ", element);
31         stack.push(element); // push element onto stack
32     }
33 }
34
```

Fig. 20.9 | Passing generic Stack objects to generic methods. (Part 2 of 4.)

```
35 // generic method testPop pops elements from a Stack
36 public static <E> void testPop(String name, Stack<E> stack) {
37     // pop elements from stack
38     try {
39         System.out.printf("%nPopping elements from %s%n", name);
40         E popValue; // store element removed from stack
41
42         // remove all elements from Stack
43         while (true) {
44             popValue = stack.pop();
45             System.out.printf("%s ", popValue);
46         }
47     }
48     catch(NoSuchElementException noSuchElementException) {
49         System.out.println();
50         noSuchElementException.printStackTrace();
51     }
52 }
53 }
```

Fig. 20.9 | Passing generic Stack objects to generic methods. (Part 3 of 4.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest2.testPop(StackTest2.java:44)
    at StackTest2.main(StackTest2.java:16)
```

```
Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest2.testPop(StackTest2.java:44)
    at StackTest2.main(StackTest2.java:20)
```

Fig. 20.9 | Passing generic Stack objects to generic methods. (Part 4 of 4.)

```
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 Numbers containing
9         // both Integers and Doubles, then display total of the elements
10        Number[] numbers = {1, 2.4, 3, 4.1}; // Integers and Doubles
11        List<Number> numberList = new ArrayList<>();
12
13        for (Number element : numbers) {
14            numberList.add(element); // place each number in numberList
15        }
16
17        System.out.printf("numberList contains: %s%n", numberList);
18        System.out.printf("Total of the elements in numberList: %.1f%n",
19                         sum(numberList));
20    }
}
```

Fig. 20.10 | Totaling the numbers in a List<Number>.

```
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>.

```
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 Integers, then
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}
```

Fig. 20.11 | Wildcard test program. (Part I of 4.)

```
18     System.out.printf("integerList contains: %s%n", integerList);
19     System.out.printf("Total of the elements in integerList: %.0f%n%n",
20                       sum(integerList));
21
22     // create, initialize and output List of Doubles, then
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%n", doubleList);
33     System.out.printf("Total of the elements in doubleList: %.1f%n%n",
34                       sum(doubleList));
35
```

Fig. 20.11 | Wildcard test program. (Part 2 of 4.)

```
36     // create, initialize and output List of Numbers containing
37     // both Integers and Doubles, then display total of the elements
38     Number[] numbers = {1, 2.4, 3, 4.1}; // Integers and Doubles
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%n", numberList);
47     System.out.printf("Total of the elements in numberList: %.1f%n",
48                     sum(numberList));
49 }
50
```

Fig. 20.11 | Wildcard test program. (Part 3 of 4.)

```
51     // total the elements; using a wildcard in the List parameter
52     public static double sum(List<? extends Number> list) {
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. (Part 4 of 4.)



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.