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

```java
// Fig. 20.1: OverloadedMethods.java
// Printing array elements using overloaded methods.

public class OverloadedMethods {
   public static void main(String[] args) {
      // create arrays of Integer, Double and Character
      Integer[] integerArray = {1, 2, 3, 4, 5, 6};
      Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
      Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};

      System.out.printf("Array integerArray contains: ");
      printArray(integerArray); // pass an Integer array
      System.out.printf("Array doubleArray contains: ");
      printArray(doubleArray); // pass a Double array
      System.out.printf("Array characterArray contains: ");
      printArray(characterArray); // pass a Character array
   }
```

Fig. 20.1 | Printing array elements using overloaded methods. (Part 1 of 3.)

```
   // method printArray to print Integer array
   public static void printArray(Integer[] inputArray) {
      // display array elements
      for (Integer element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }

   // method printArray to print Double array
   public static void printArray(Double[] inputArray) {
      // display array elements
      for (Double element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }
```

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

```java

   // method printArray to print Character array
   public static void printArray(Character[] inputArray) {
      // display array elements
      for (Character element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }
}
```

```
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.)

```java
public static void printArray(T[] inputArray) {
   // display array elements
   for (T element : inputArray)  {
      System.out.printf("%s ", element);
   }

   System.out.println();
}
```

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

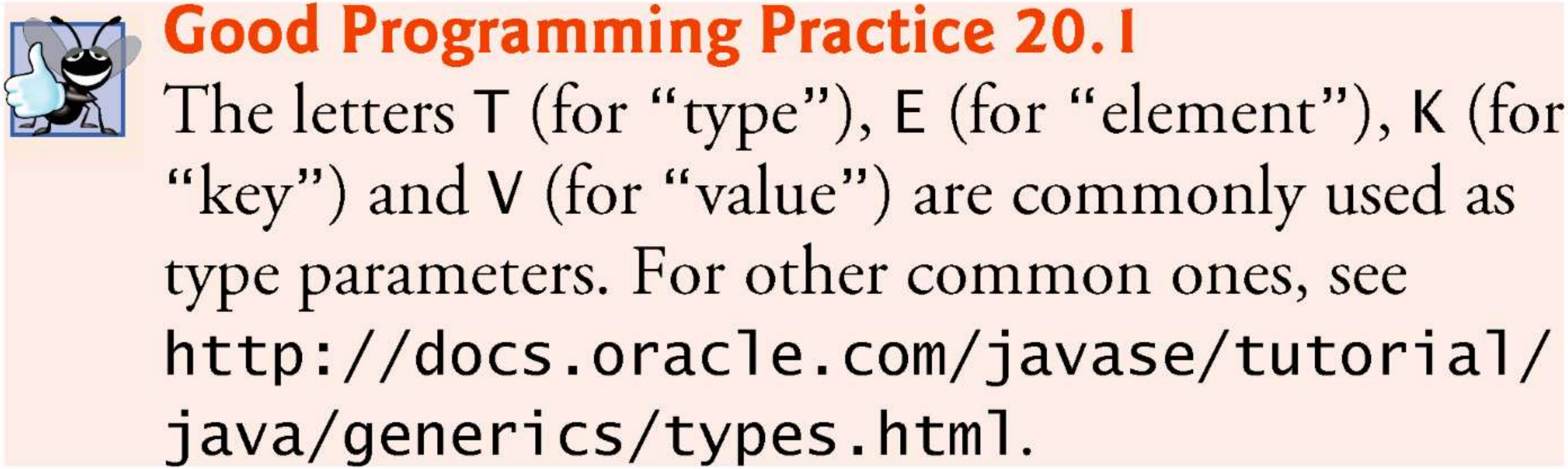
```java
// Fig. 20.3: GenericMethodTest.java
// Printing array elements using generic method printArray.

public class GenericMethodTest {
   public static void main(String[] args) {
      // create arrays of Integer, Double and Character
      Integer[] integerArray = {1, 2, 3, 4, 5};
      Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
      Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};

      System.out.printf("Array integerArray contains: ");
      printArray(integerArray); // pass an Integer array
      System.out.printf("Array doubleArray contains: ");
      printArray(doubleArray); // pass a Double array
      System.out.printf("Array characterArray contains: ");
      printArray(characterArray); // pass a Character array
   }
```

Fig. 20.3 | Printing array elements using generic method `printArray`. (Part 1 of 2.)

```java

   // generic method printArray
   public static <T> void printArray(T[] inputArray) {
      // display array elements
      for (T element : inputArray) {
         System.out.printf("%s ", element);
      }

      System.out.println();
   }
}
```

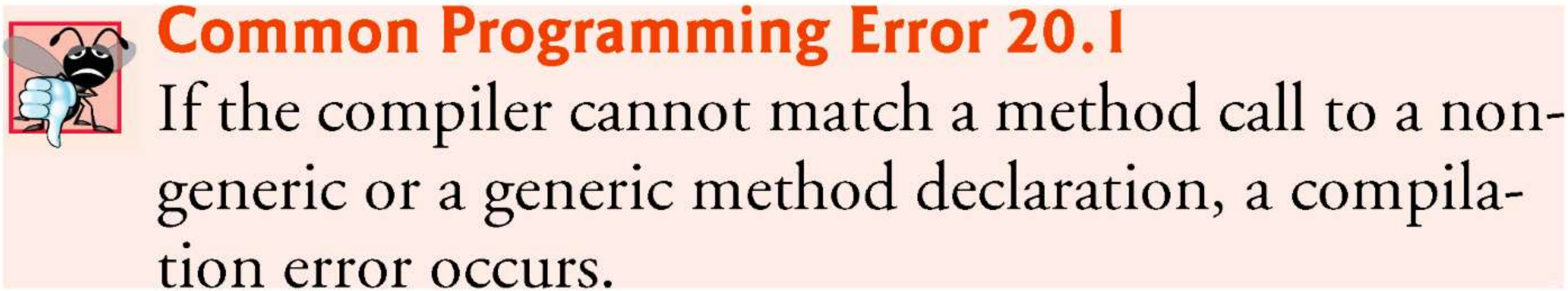
```
Array integerArray contains: 1 2 3 4 5
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array characterArray contains: H E L L O
```

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



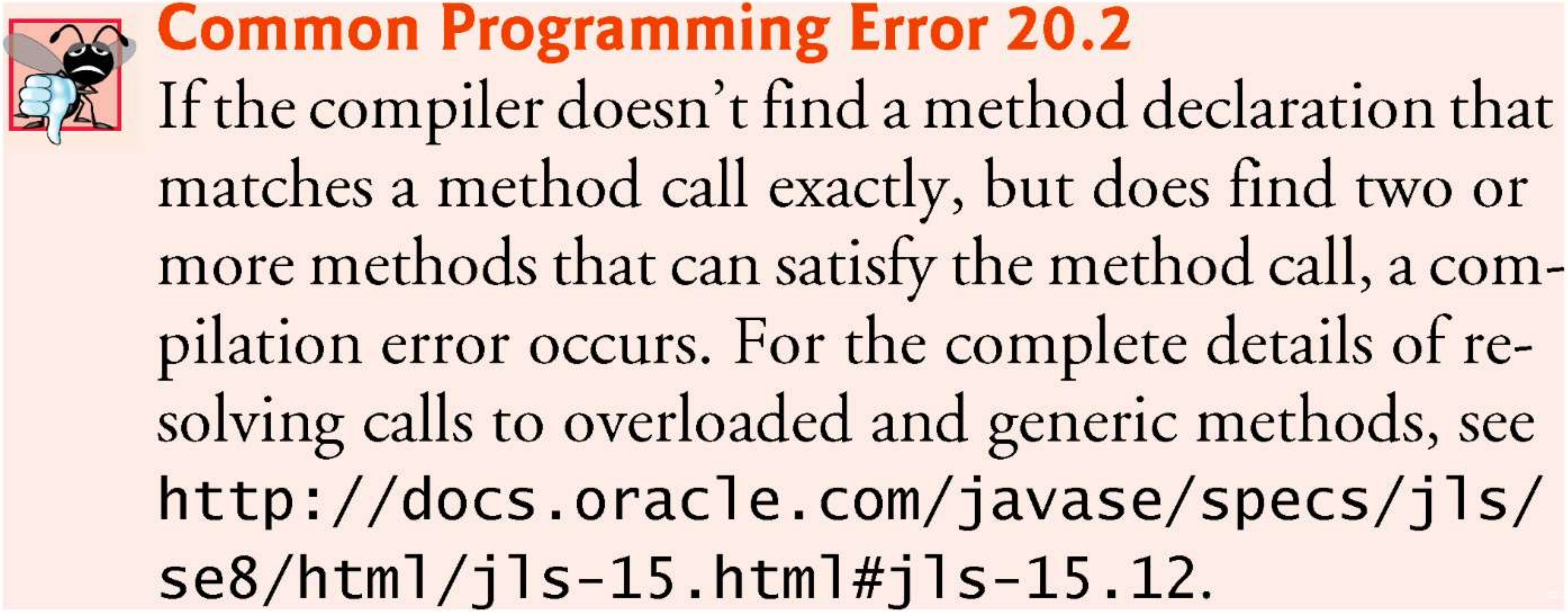
Good Programming Practice 20.1

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`.



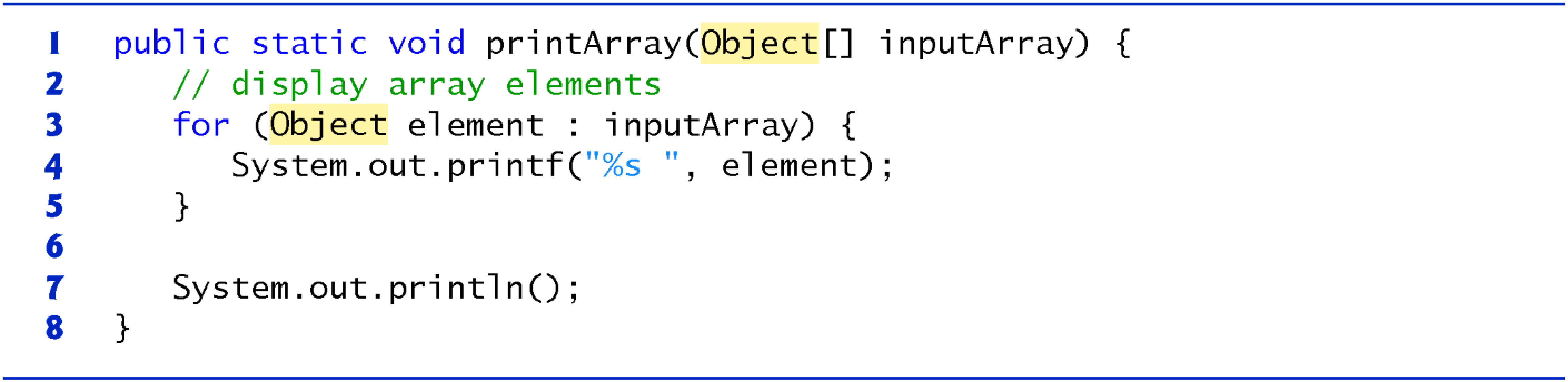
Common Programming Error 20.1

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`.



```java
public static void printArray(Object[] inputArray) {
   // display array elements
   for (Object element : inputArray) {
      System.out.printf("%s ", element);
   }

   System.out.println();
}
```

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

```java
// Fig. 20.5: MaximumTest.java
// Generic method maximum returns the largest of three objects.

public class MaximumTest {
   public static void main(String[] args) {
      System.out.printf("Maximum of %d, %d and %d is %d%n", 3, 4, 5,
         maximum(3, 4, 5));
      System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f%n",
         6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
      System.out.printf("Maximum of %s, %s and %s is %s%n", "pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }

```

Fig. 20.5 | Generic method `maximum` with an upper bound on its type parameter. (Part 1 of 2.)

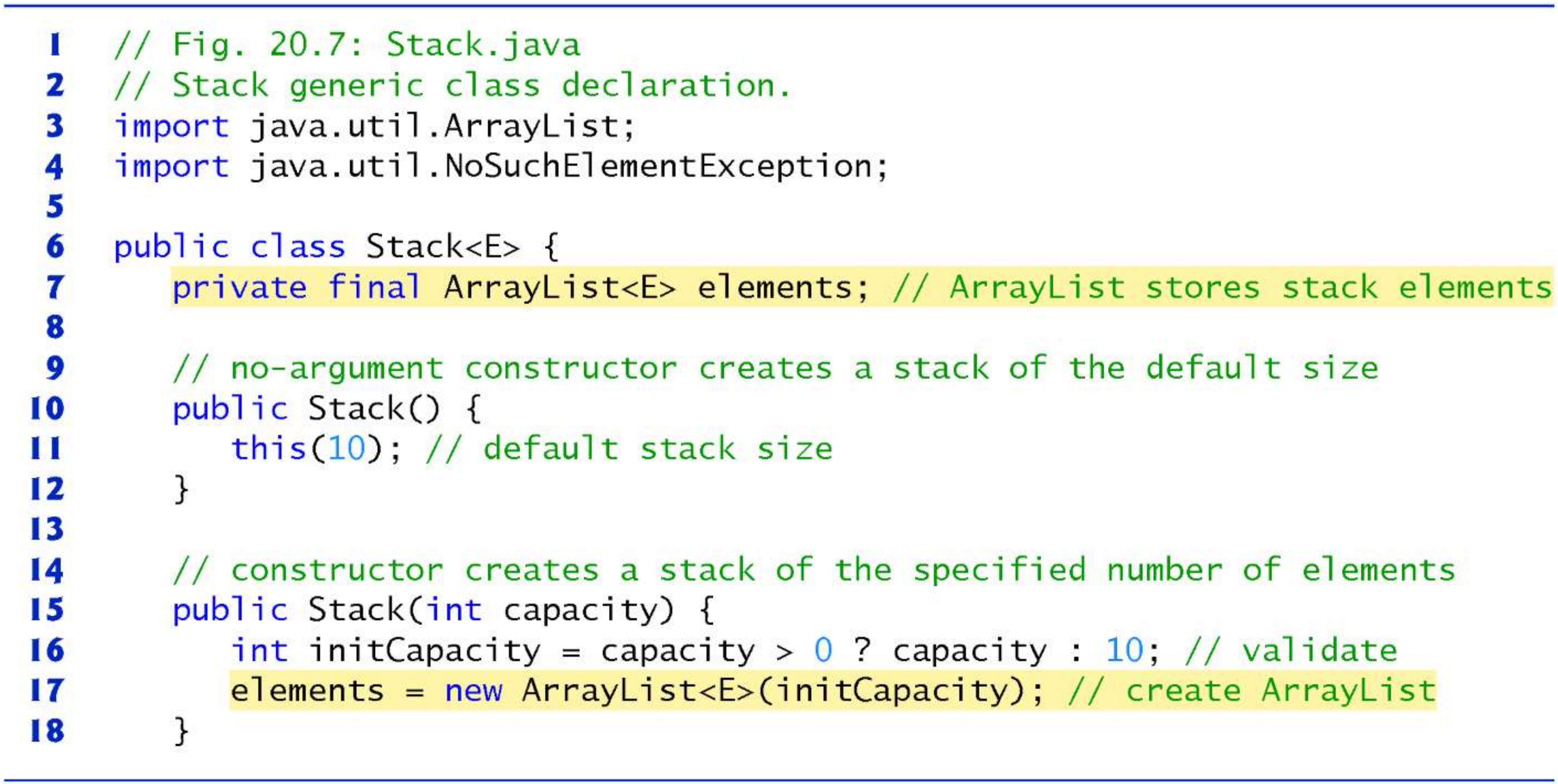
```java
   // determines the largest of three Comparable objects
   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x; // assume x is initially the largest

      if (y.compareTo(max) > 0) {
         max = y; // y is the largest so far
      }

      if (z.compareTo(max) > 0) {
         max = z; // z is the largest
      }

      return max; // returns the largest object
   }
}
```

```
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.)

```java
public static Comparable maximum(Comparable x, Comparable y,
   Comparable z) {

   Comparable max = x; // assume x is initially the largest

   if (y.compareTo(max) > 0) {
      max = y; // y is the largest so far
   }

   if (z.compareTo(max) > 0) {
      max = z; // z is the largest
   }

   return max; // returns the largest object
}
```

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



```java
// Fig. 20.7: Stack.java
// Stack generic class declaration.
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class Stack<E> {
   private final ArrayList<E> elements; // ArrayList stores stack elements

   // no-argument constructor creates a stack of the default size
   public Stack() {
      this(10); // default stack size
   }

   // constructor creates a stack of the specified number of elements
   public Stack(int capacity) {
      int initCapacity = capacity > 0 ? capacity : 10; // validate
      elements = new ArrayList<E>(initCapacity); // create ArrayList
   }
```

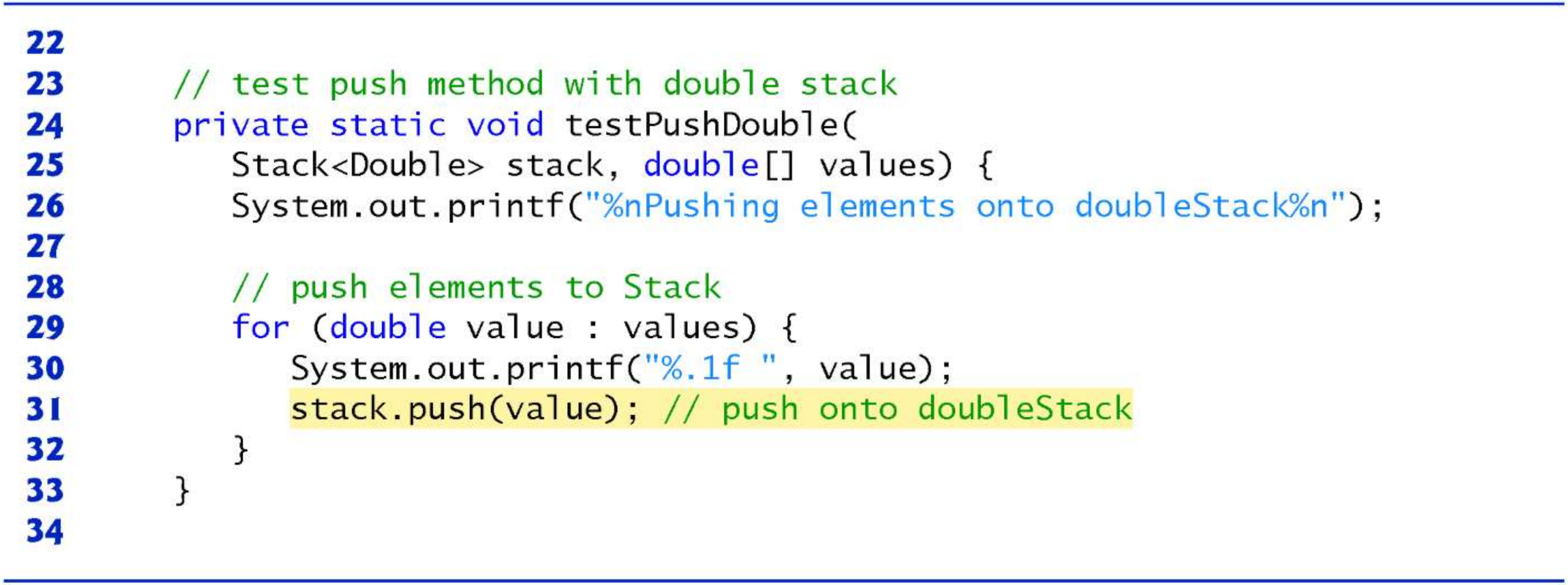
Fig. 20.7 | Stack generic class declaration. (Part 1 of 2.)

```java

   // push element onto stack
   public void push(E pushValue) {
      elements.add(pushValue); // place pushValue on Stack
   }

   // return the top element if not empty; else throw exception
   public E pop() {
      if (elements.isEmpty()) { // if stack is empty
         throw new NoSuchElementException("Stack is empty, cannot pop");
      }

      // remove and return top element of Stack
      return elements.remove(elements.size() - 1);
   }
}
```

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

```java
// Fig. 20.8: StackTest.java
// Stack generic class test program.
import java.util.NoSuchElementException;

public class StackTest {
   public static void main(String[] args) {
      double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
      int[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

      // Create a Stack<Double> and a Stack<Integer>
      Stack<Double> doubleStack = new Stack<>(5);
      Stack<Integer> integerStack = new Stack<>();

      // push elements of doubleElements onto doubleStack
      testPushDouble(doubleStack, doubleElements);
      testPopDouble(doubleStack); // pop from doubleStack

      // push elements of integerElements onto integerStack
      testPushInteger(integerStack, integerElements);
      testPopInteger(integerStack); // pop from integerStack
   }
```

Fig. 20.8 | Stack generic class test program. (Part 1 of 6.)



```java

    // test push method with double stack
    private static void testPushDouble(
       Stack<Double> stack, double[] values) {
       System.out.printf("%nPushing elements onto doubleStack%n");

       // push elements to Stack
       for (double value : values) {
          System.out.printf("%.1f ", value);
          stack.push(value); // push onto doubleStack
       }
    }

```

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

```java
   // test pop method with double stack
   private static void testPopDouble(Stack<Double> stack) {
      // pop elements from stack
      try {
         System.out.printf("%nPopping elements from doubleStack%n");
         double popValue; // store element removed from stack

         // remove all elements from Stack
         while (true) {
            popValue = stack.pop(); // pop from doubleStack
            System.out.printf("%.1f ", popValue);
         }
      }
      catch(NoSuchElementException noSuchElementException) {
         System.err.println();
         noSuchElementException.printStackTrace();
      }
   }
```

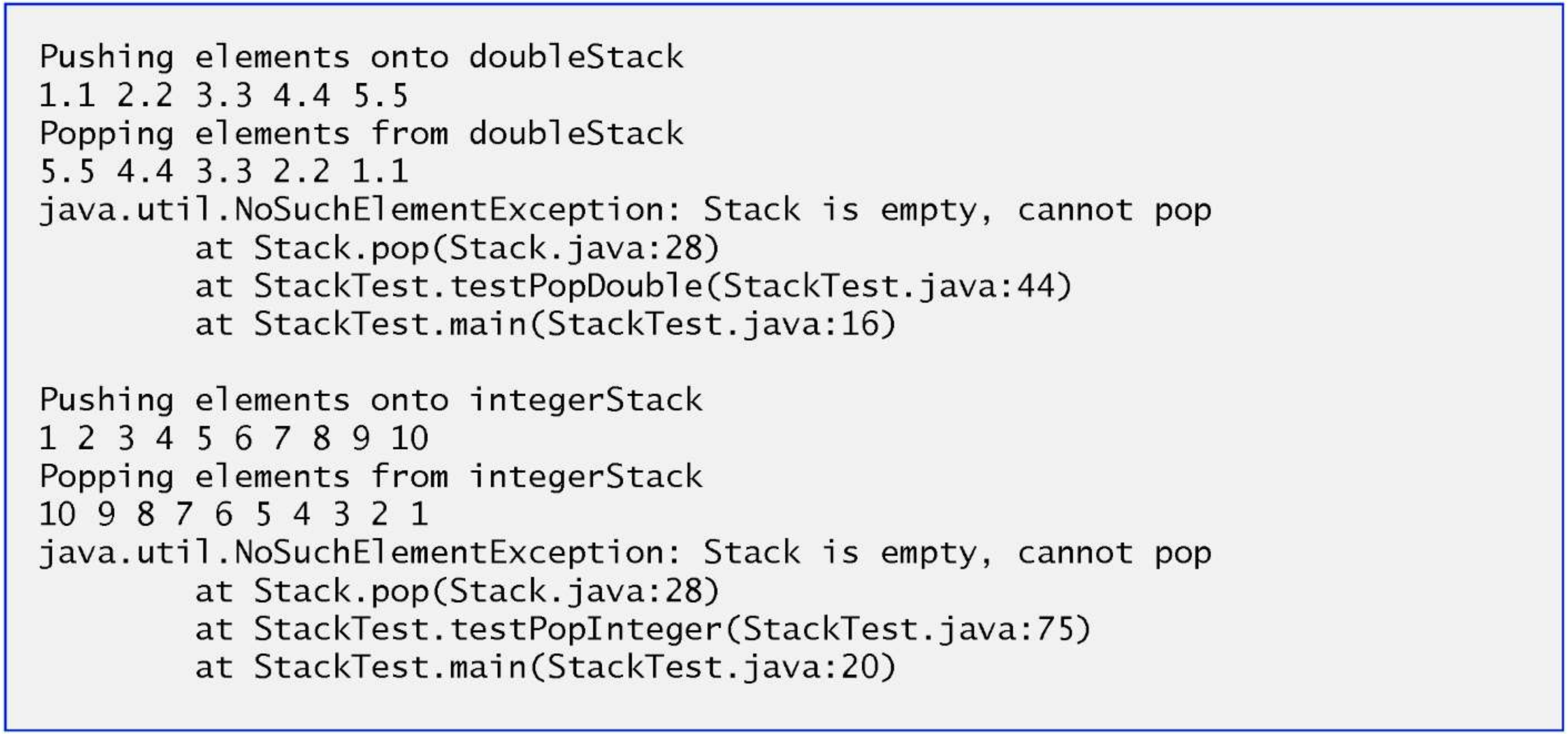
Fig. 20.8 | Stack generic class test program. (Part 3 of 6.)

```java

   // test push method with integer stack
   private static void testPushInteger(
      Stack<Integer> stack, int[] values) {
      System.out.printf("%nPushing elements onto integerStack%n");

      // push elements to Stack
      for (int value : values) {
         System.out.printf("%d ", value);
         stack.push(value); // push onto integerStack
      }
   }

```

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

```
   // test pop method with integer stack
   private static void testPopInteger(Stack<Integer> stack) {
      // pop elements from stack
      try {
         System.out.printf("%nPopping elements from integerStack%n");
         int popValue; // store element removed from stack

         // remove all elements from Stack
         while (true) {
            popValue = stack.pop(); // pop from intStack
            System.out.printf("%d ", popValue);
         }
      }
      catch(NoSuchElementException noSuchElementException) {
         System.err.println();
         noSuchElementException.printStackTrace();
      }
   }
}
```

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.)

```java
// Fig. 20.9: StackTest2.java
// Passing generic Stack objects to generic methods.
import java.util.NoSuchElementException;

public class StackTest2 {
   public static void main(String[] args) {
      Double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
      Integer[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

      // Create a Stack<Double> and a Stack<Integer>
      Stack<Double> doubleStack = new Stack<>(5);
      Stack<Integer> integerStack = new Stack<>();

      // push elements of doubleElements onto doubleStack
      testPush("doubleStack", doubleStack, doubleElements);
      testPop("doubleStack", doubleStack); // pop from doubleStack

      // push elements of integerElements onto integerStack
      testPush("integerStack", integerStack, integerElements);
      testPop("integerStack", integerStack); // pop from integerStack
   }
```

Fig. 20.9 | Passing generic Stack objects to generic methods. (Part 1 of 4.)

```java

   // generic method testPush pushes elements onto a Stack
   public static <E> void testPush(String name , Stack<E> stack,
      E[] elements) {
      System.out.printf("%nPushing elements onto %s%n", name);

      // push elements onto Stack
      for (E element : elements) {
         System.out.printf("%s ", element);
         stack.push(element); // push element onto stack
      }
   }

```

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

```java
   // generic method testPop pops elements from a Stack
   public static <E> void testPop(String name, Stack<E> stack) {
      // pop elements from stack
      try {
         System.out.printf("%nPopping elements from %s%n", name);
         E popValue; // store element removed from stack

         // remove all elements from Stack
         while (true) {
            popValue = stack.pop();
            System.out.printf("%s ", popValue);
         }
      }
      catch(NoSuchElementException noSuchElementException) {
         System.out.println();
         noSuchElementException.printStackTrace();
      }
   }
}
```

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.)

```java
// Fig. 20.10: TotalNumbers.java
// Totaling the numbers in a List<Number>.
import java.util.ArrayList;
import java.util.List;

public class TotalNumbers {
   public static void main(String[] args) {
      // create, initialize and output List of Numbers containing
      // both Integers and Doubles, then display total of the elements
      Number[] numbers = {1, 2.4, 3, 4.1}; // Integers and Doubles
      List<Number> numberList = new ArrayList<>();

      for (Number element : numbers) {
         numberList.add(element); // place each number in numberList
      }

      System.out.printf("numberList contains: %s%n", numberList);
      System.out.printf("Total of the elements in numberList: %.1f%n",
         sum(numberList));
   }
```

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

```java

   // calculate total of List elements
   public static double sum(List<Number> list) {
      double total = 0; // initialize total

      // calculate sum
      for (Number element : list) {
         total += element.doubleValue();
      }

      return total;
   }
}
```

```
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>`.

```java
// Fig. 20.11: WildcardTest.java
// Wildcard test program.
import java.util.ArrayList;
import java.util.List;

public class WildcardTest {
   public static void main(String[] args) {
      // create, initialize and output List of Integers, then
      // display total of the elements
      Integer[] integers = {1, 2, 3, 4, 5};
      List<Integer> integerList = new ArrayList<>();

      // insert elements in integerList
      for (Integer element : integers) {
         integerList.add(element);
      }

```

Fig. 20.11 | Wildcard test program. (Part 1 of 4.)

```java
      System.out.printf("integerList contains: %s%n", integerList);
      System.out.printf("Total of the elements in integerList: %.0f%n%n",
         sum(integerList));

      // create, initialize and output List of Doubles, then
      // display total of the elements
      Double[] doubles = {1.1, 3.3, 5.5};
      List<Double> doubleList = new ArrayList<>();

      // insert elements in doubleList
      for (Double element : doubles) {
         doubleList.add(element);
      }

      System.out.printf("doubleList contains: %s%n", doubleList);
      System.out.printf("Total of the elements in doubleList: %.1f%n%n",
         sum(doubleList));

```

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

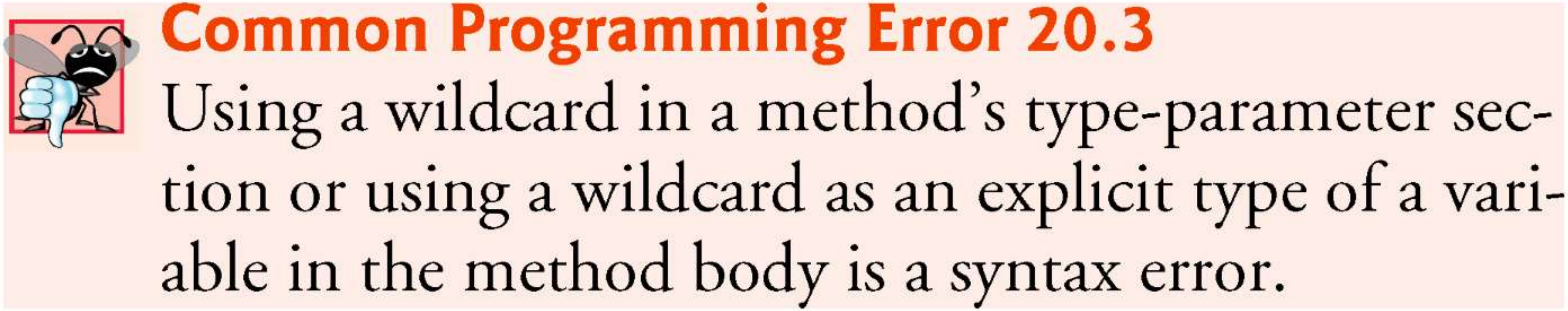
```
      // create, initialize and output List of Numbers containing
      // both Integers and Doubles, then display total of the elements
      Number[] numbers = {1, 2.4, 3, 4.1}; // Integers and Doubles
      List<Number> numberList = new ArrayList<>();

      // insert elements in numberList
      for (Number element : numbers) {
         numberList.add(element);
      }

      System.out.printf("numberList contains: %s%n", numberList);
      System.out.printf("Total of the elements in numberList: %.1f%n",
         sum(numberList));
   }

```

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

```java
   // total the elements; using a wildcard in the List parameter
   public static double sum(List<? extends Number> list) {
      double total = 0; // initialize total

      // calculate sum
      for (Number element : list) {
         total += element.doubleValue();
      }

      return total;
   }
}
```

```
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.