Java Programming

Generic classes and methods

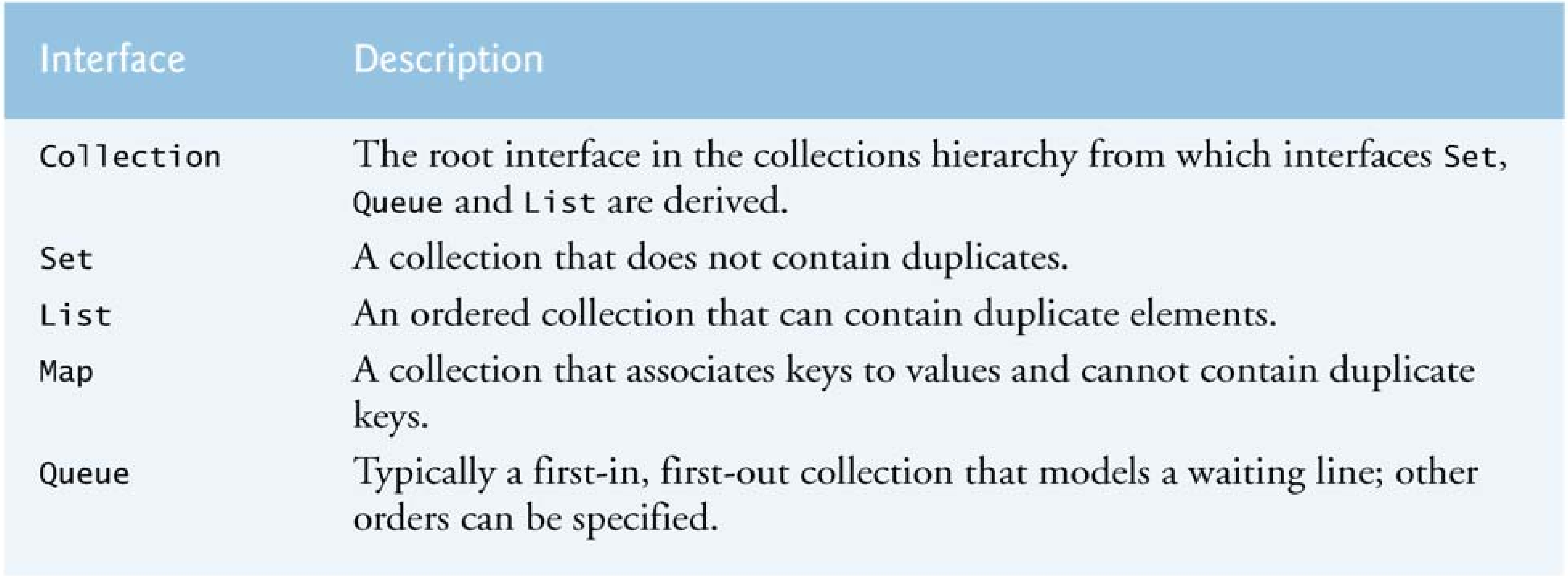
Object Oriented Programming

- Variables: local, Instance, `static, final`
- Methods: setters, getters, constructors, static, method overloading
- this Keyword, Random numbers
- `Arrays: One/ Multiple dimensional arrays`
  - `Array class`
  - `ArrayList`
- Enumerations
- Composition and Inheritance
- Polymorphism
- Interfaces
- Generic classes and methods

Generic Collection

- Java collections framework
  - prebuilt data structures
  - interfaces and methods for manipulating those data structures

- A collection is a data structure—actually, an object—that can hold references to other objects.
  - Usually, collections contain references to objects that are all of the same type.



Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does not contain duplicates.
`List`	An ordered collection that can contain duplicate elements.
`Map`	A collection that associates keys to values and cannot contain duplicate keys.
`Queue`	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Fig. 20.1 | Some collections-framework interfaces.

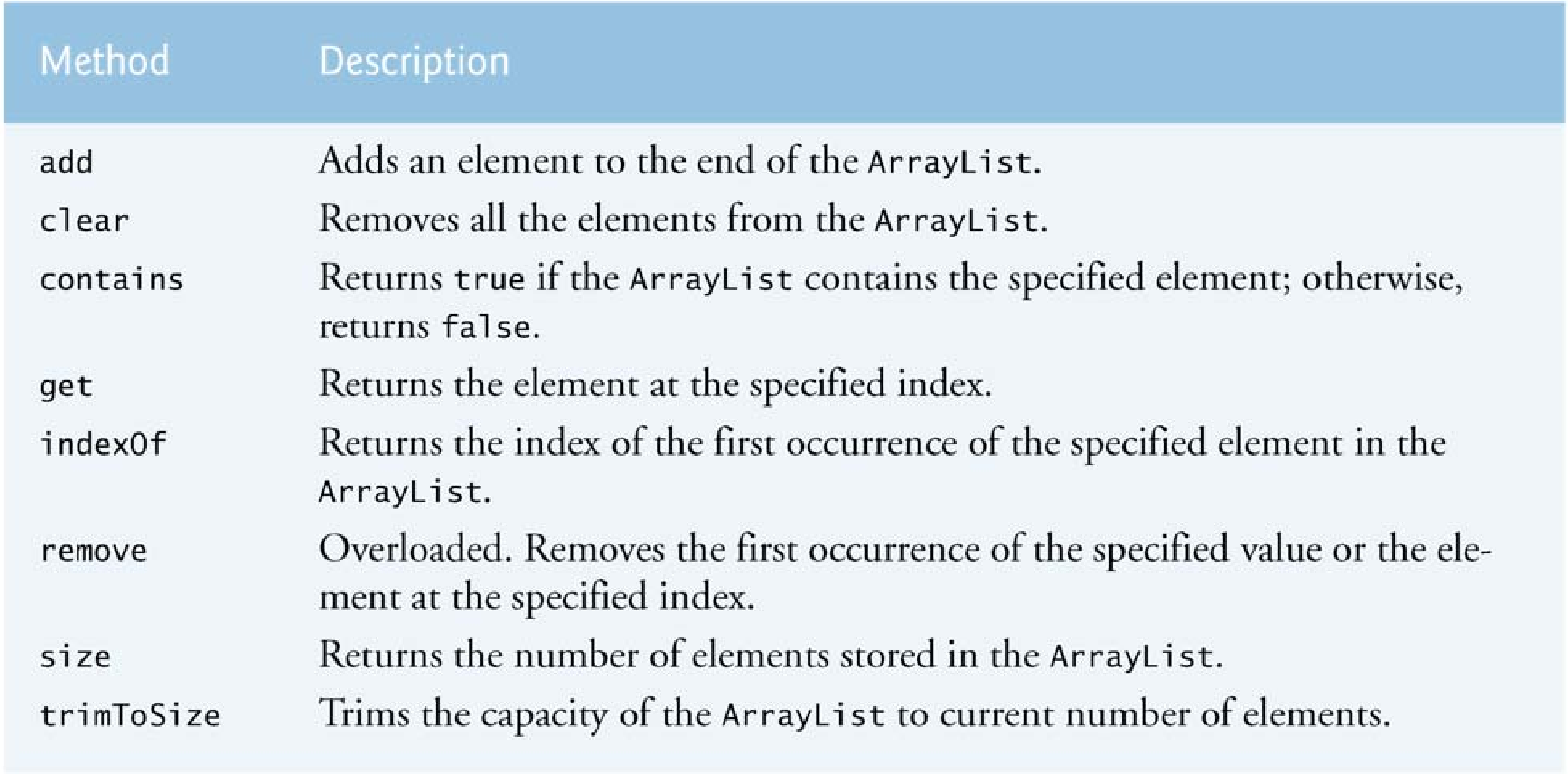
Class `ArrayList`

- Arrays do not automatically change their size at execution time to accommodate additional elements.

- `ArrayList<T>` (package `java.util`) can dynamically change its size to accommodate more elements.
  - `T` is a placeholder for the type of element stored in the collection.
  - This is similar to specifying the type when declaring an array, except that only nonprimitive types can be used with these collection classes.

- Classes with this kind of placeholder that can be used with any type are called generic classes.

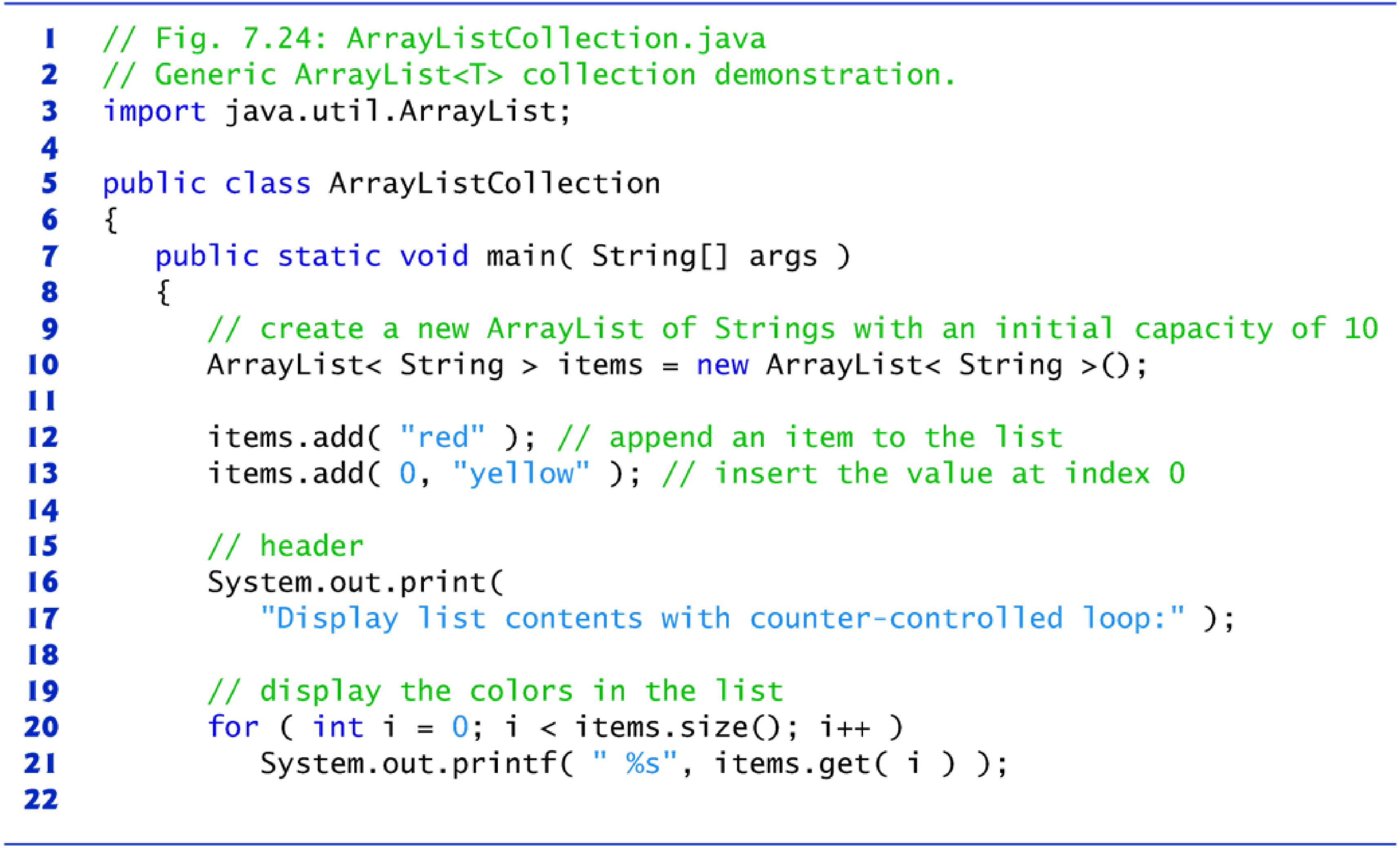
Class ArrayList



Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

Class ArrayList



```java
// Fig. 7.24: ArrayListCollection.java
// Generic ArrayList<T> collection demonstration.
import java.util.ArrayList;

public class ArrayListCollection
{
   public static void main( String[] args )
   {
      // create a new ArrayList of Strings with an initial capacity of 10
      ArrayList< String > items = new ArrayList< String >();

      items.add( "red" ); // append an item to the list
      items.add( 0, "yellow" ); // insert the value at index 0

      // header
      System.out.print(
         "Display list contents with counter-controlled loop:" );

      // display the colors in the list
      for ( int i = 0; i < items.size(); i++ )
         System.out.printf( " %s", items.get( i ) );

```

Fig. 7.24 | Generic ArrayList<T> collection demonstration. (Part 1 of 3.)

Class ArrayList

```java
      // display colors using foreach in the display method
      display( items,
         "\nDisplay list contents with enhanced for statement:" );

      items.add( "green" ); // add "green" to the end of the list
      items.add( "yellow" ); // add "yellow" to the end of the list
      display( items, "List with two new elements:" );

      items.remove( "yellow" ); // remove the first "yellow"
      display( items, "Remove first instance of yellow:" );

      items.remove( 1 ); // remove item at index 1
      display( items, "Remove second list element (green):" );

      // check if a value is in the List
      System.out.printf( "\"red\" is %sin the list\n",
         items.contains( "red" ) ? "": "not " );

      // display number of elements in the List
      System.out.printf( "Size: %s\n", items.size() );
   } // end main

```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 2 of 3.)

Class ArrayList

```java
   // display the ArrayList's elements on the console
   public static void display( ArrayList< String > items, String header )
   {
      System.out.print( header ); // display header

      // display each element in items
      for ( String item : items )
         System.out.printf( " %s", item );

      System.out.println(); // display end of line
   } // end method display
} // end class ArrayListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 3 of 3.)

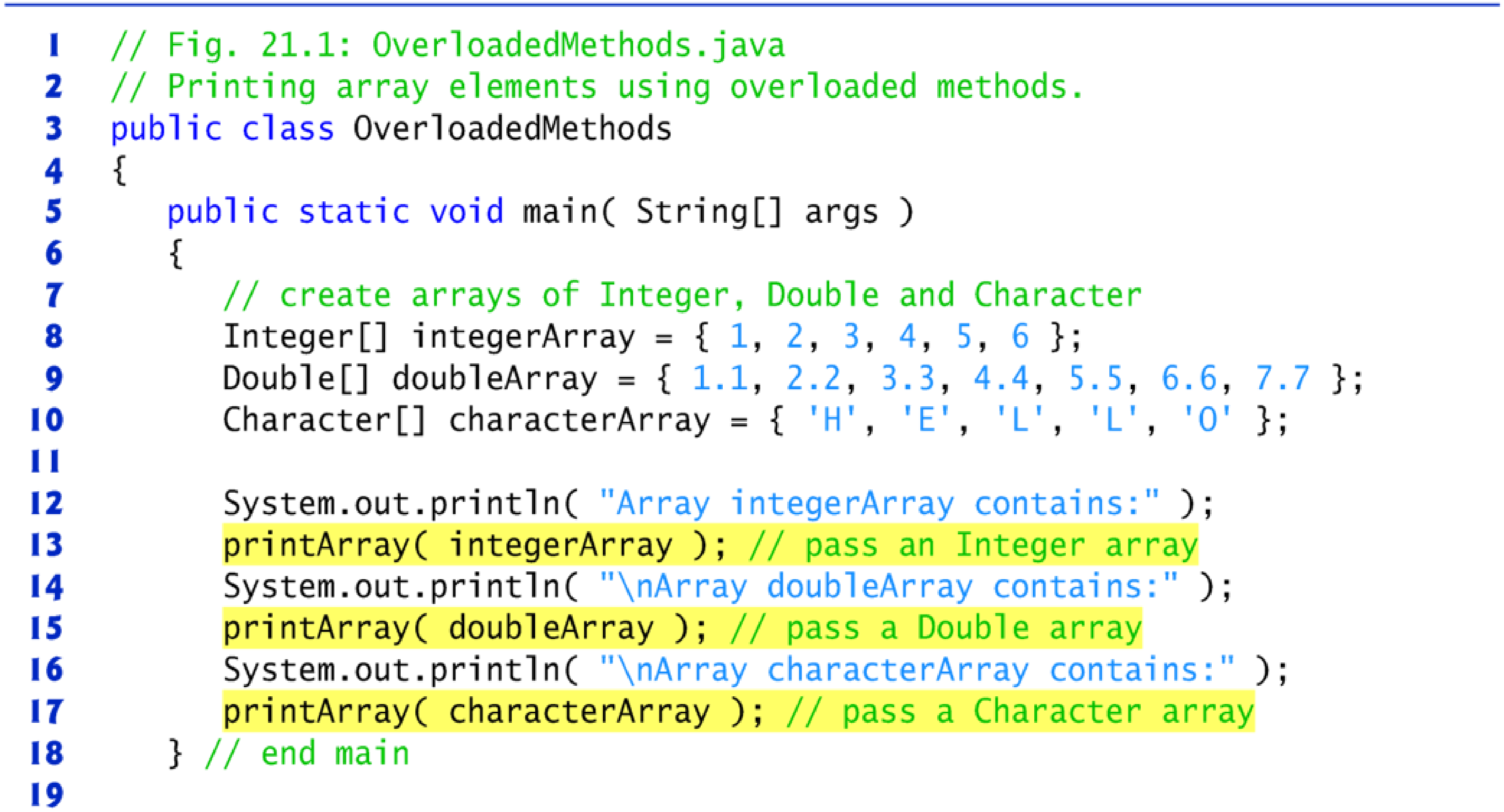
Generic Methods Generic Classes

- Generic methods and generic classes (and interfaces) enable you to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

- Generics also provide compile-time type safety that allows you to catch invalid types at compile time.

- Overloaded methods are often used to perform similar operations on different types of data.

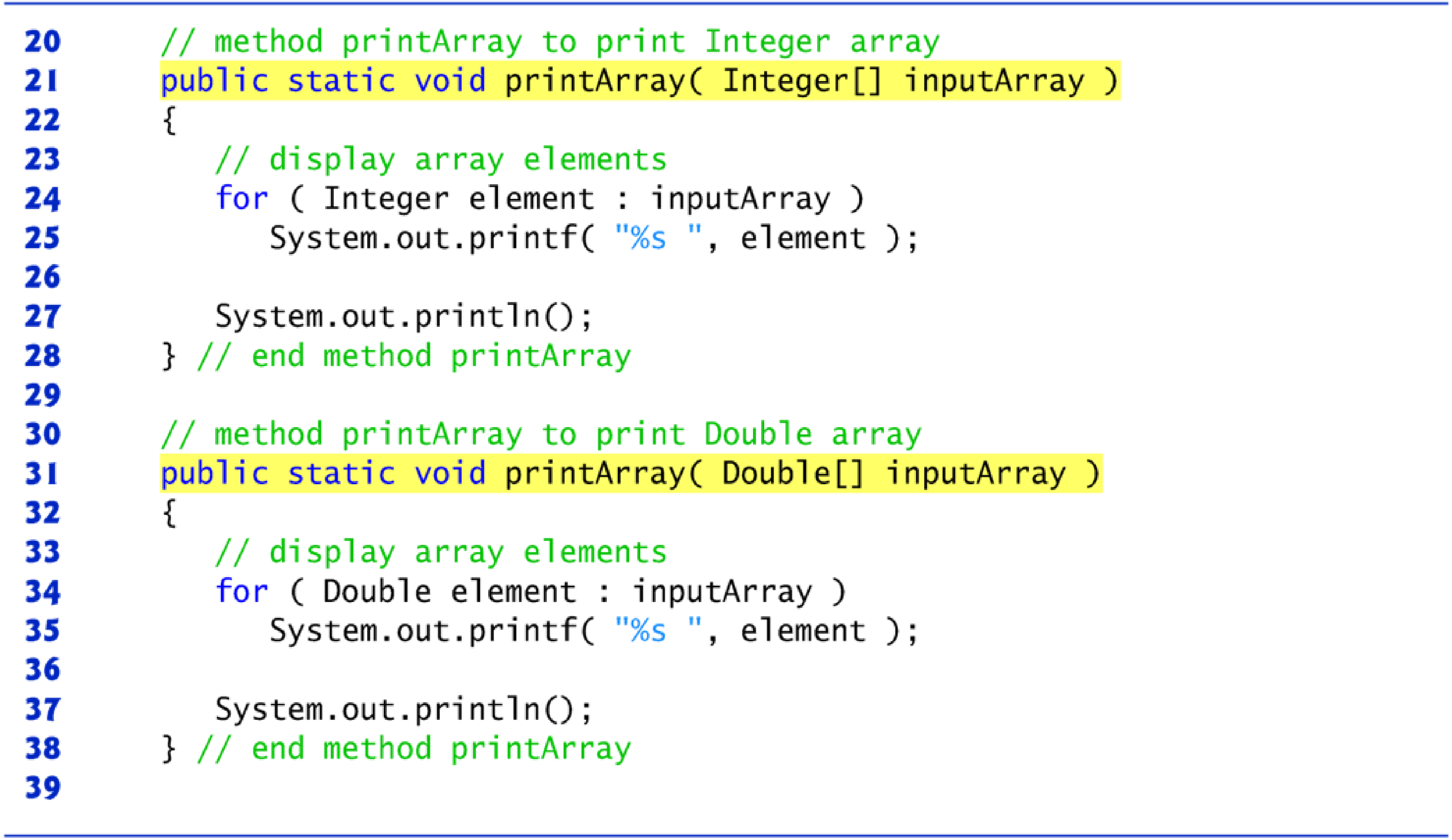
Motivation for Generic Methods



```java
// Fig. 21.1: 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.println( "Array integerArray contains:" );
      printArray( integerArray ); // pass an Integer array
      System.out.println( "\nArray doubleArray contains:" );
      printArray( doubleArray ); // pass a Double array
      System.out.println( "\nArray characterArray contains:" );
      printArray( characterArray ); // pass a Character array
   } // end main

```

Fig. 21.1 | Printing array elements using overloaded methods. (Part 1 of 3.)

Motivation for Generic Methods



```java
   // 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();
   } // end method printArray

   // 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();
   } // end method printArray

```

Fig. 21.1 | Printing array elements using overloaded methods. (Part 2 of 3.)

Motivation for Generic Methods

```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();
   } // end method printArray
} // end class OverloadedMethods
```

```
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. 21.1 | Printing array elements using overloaded methods. (Part 3 of 3.)

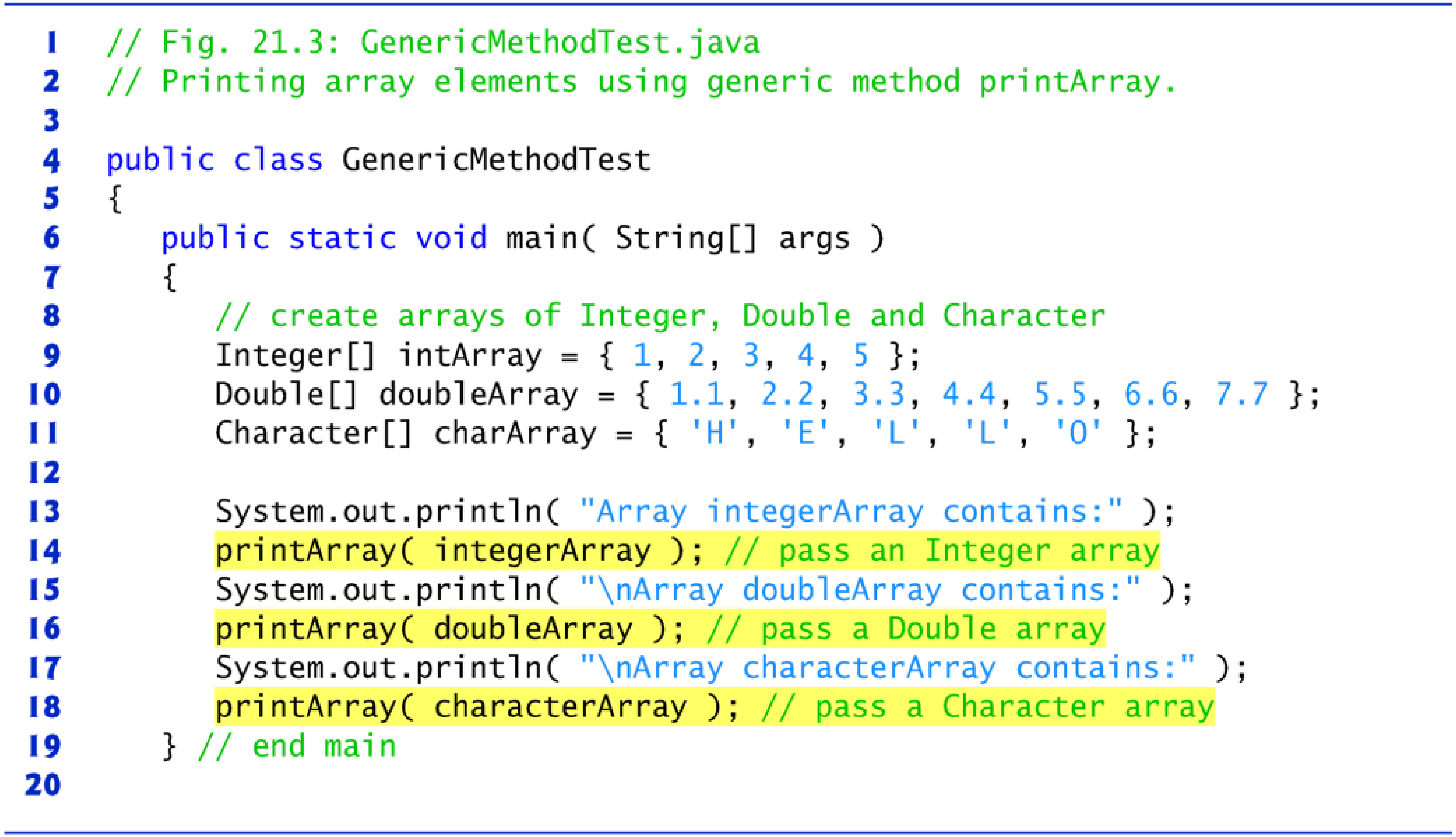
Generic Methods: Implementation and Compile-Time Translation

- If the operations performed by several overloaded methods are identical for each argument type, the overloaded methods can be more compactly and 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.

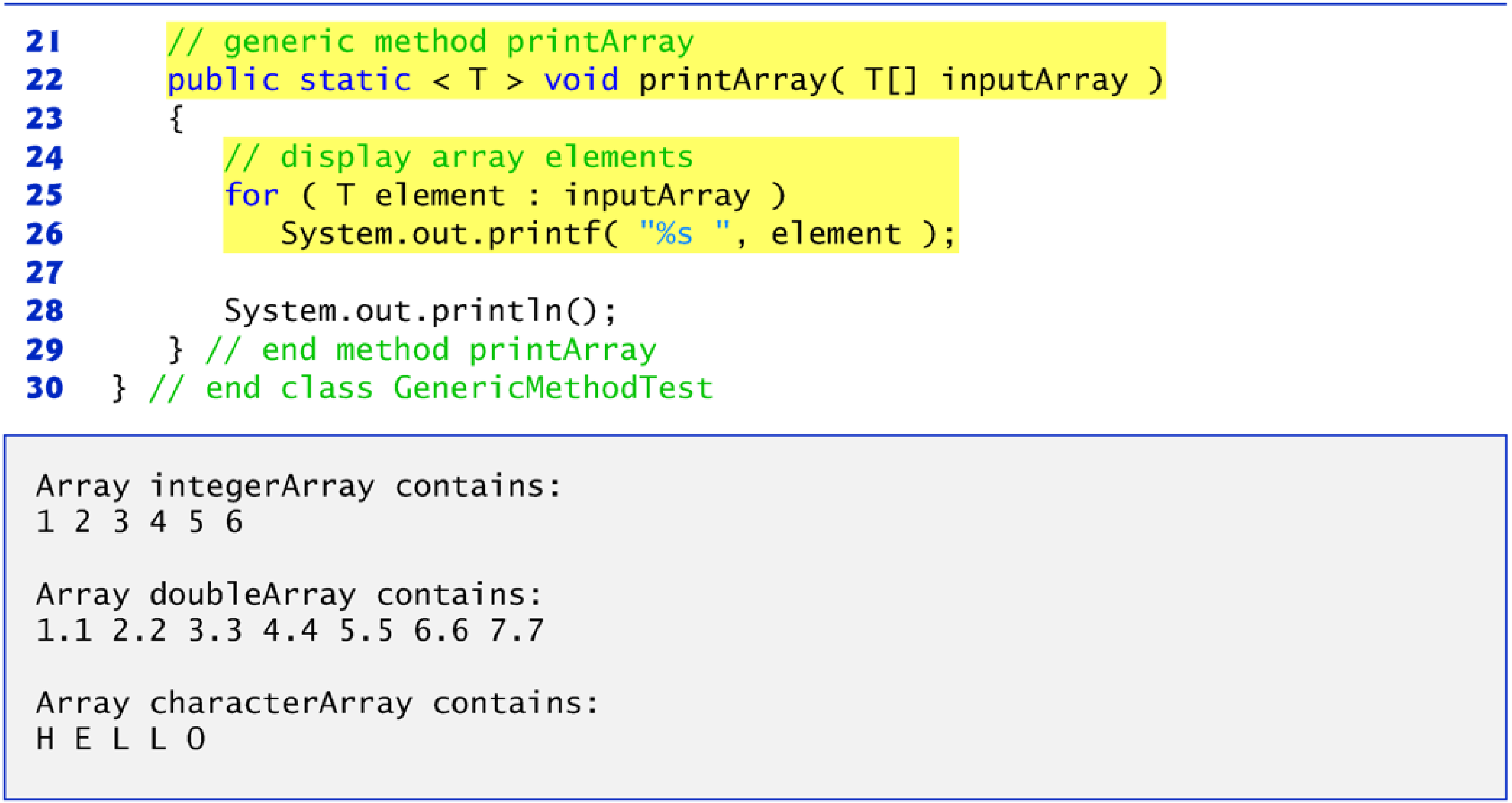
Generic Methods: Implementation and Compile-Time Translation



```java
// Fig. 21.3: 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[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println( "Array integerArray contains:" );
      printArray( integerArray ); // pass an Integer array
      System.out.println( "\nArray doubleArray contains:" );
      printArray( doubleArray ); // pass a Double array
      System.out.println( "\nArray characterArray contains:" );
      printArray( characterArray ); // pass a Character array
   } // end main

```

Fig. 21.3 | Printing array elements using generic method `printArray`. (Part 1 of 2.)

Generic Methods: Implementation and Compile-Time Translation



```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();
   } // end method printArray
} // end class GenericMethodTest
```

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

Generic Methods: Implementation and Compile-Time Translation

- All generic method declarations have a type-parameter section delimited by angle brackets (< and >) that precedes the method’s return type (< T > in this example).

- Each type-parameter section contains one or more type parameters (also called formal type parameters), separated by commas.

- A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

- Can be used to declare the return type, parameter types and local variable types in a generic method, and act as placeholders for the types of the arguments passed to the generic method (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.

Generic Methods: Implementation and Compile-Time Translation (cont.)

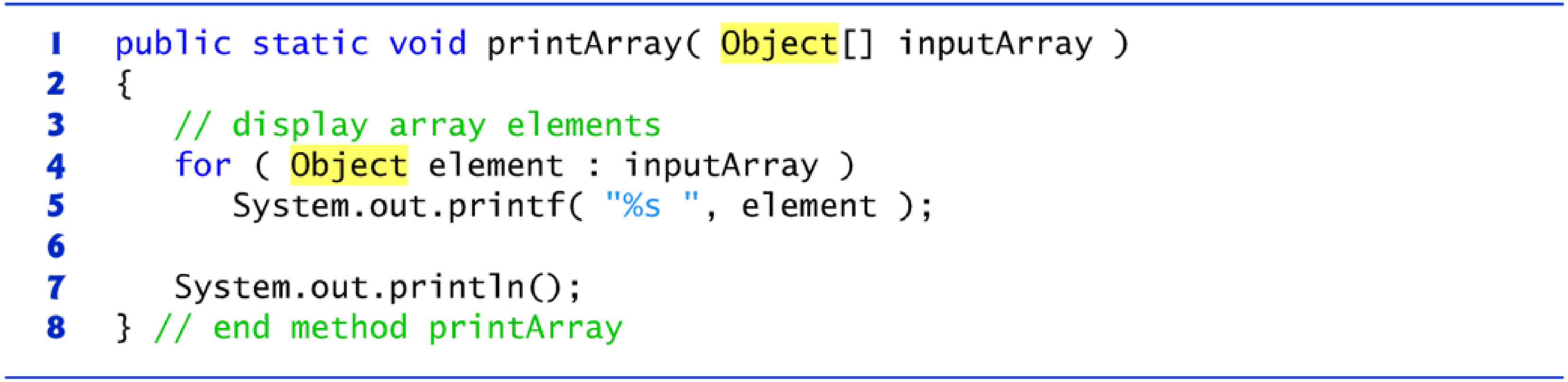
- When the compiler translates generic method `printArray` into Java bytecodes, it removes the type-parameter section and replaces the type parameters with actual types.

- 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. 21.4—there is only one copy of this code, which is used for all `printArray` calls in the example.

Generic Methods: Implementation and Compile-Time Translation (cont.)



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

   System.out.println();
} // end method printArray
```

Fig. 21.4 | Generic method `printArray` after erasure is performed by the compiler.

Overloading Generic Methods

- A generic method may be overloaded.

- A class can provide two or more generic methods that specify the same method name but different method parameters.

- A generic method can also be overloaded by nongeneric methods.

- When the compiler encounters a method call, it searches for the method declaration that most precisely matches the method name and the argument types specified in the call.

Generic Classes

- The concept of a data structure, such as a stack, can be understood independently of the element type it manipulates.

- Generic classes provide a means for describing the concept of a stack (or any other class) in a type-independent manner.

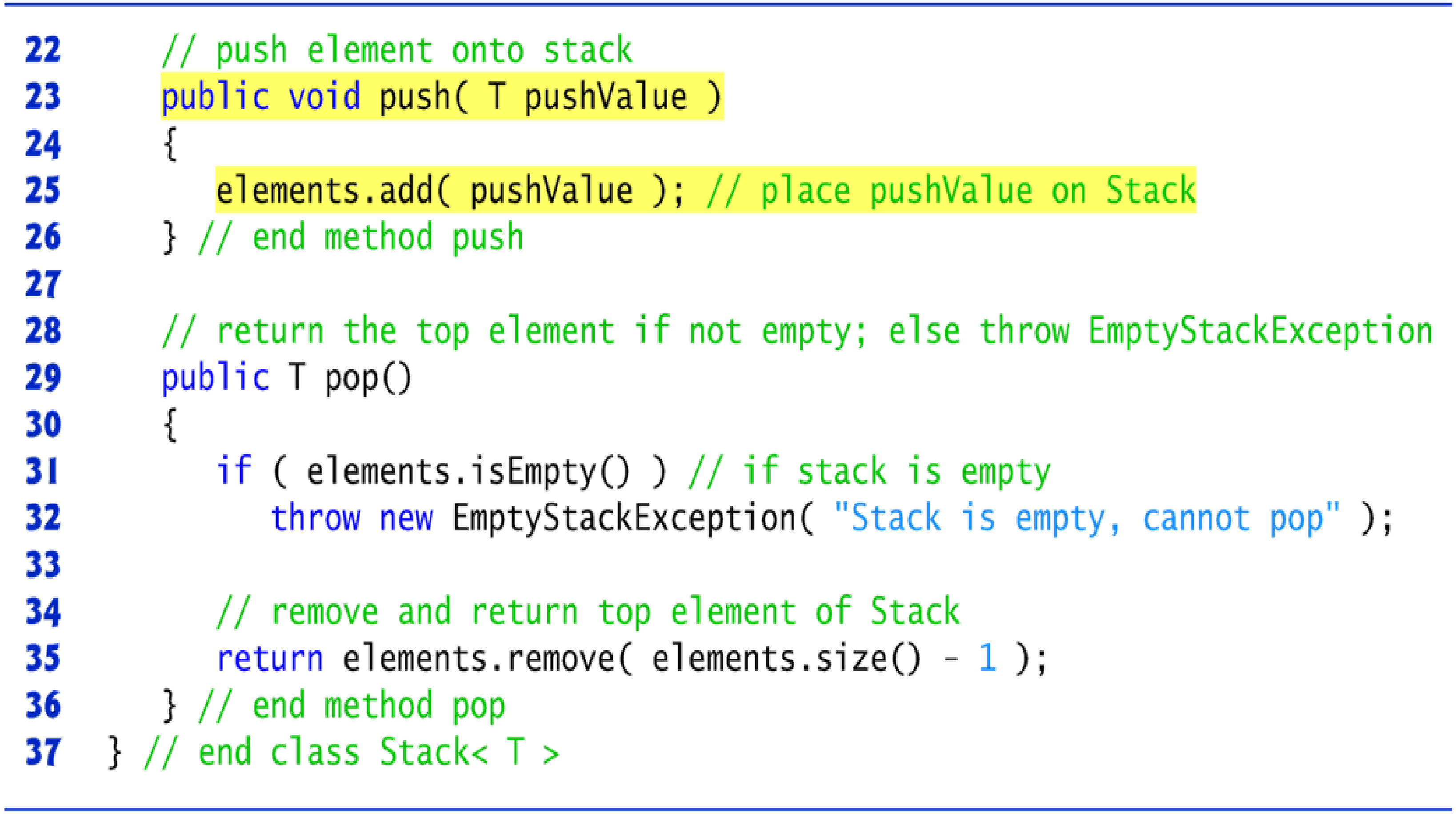
- These classes are known as parameterized classes or parameterized types because they accept one or more type parameters.

Stack Generic Class

```java
// Fig. 21.7: Stack.java
// Stack generic class declaration.
import java.util.ArrayList;

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

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

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

```

Fig. 21.7 | Stack generic class declaration. (Part 1 of 2.)

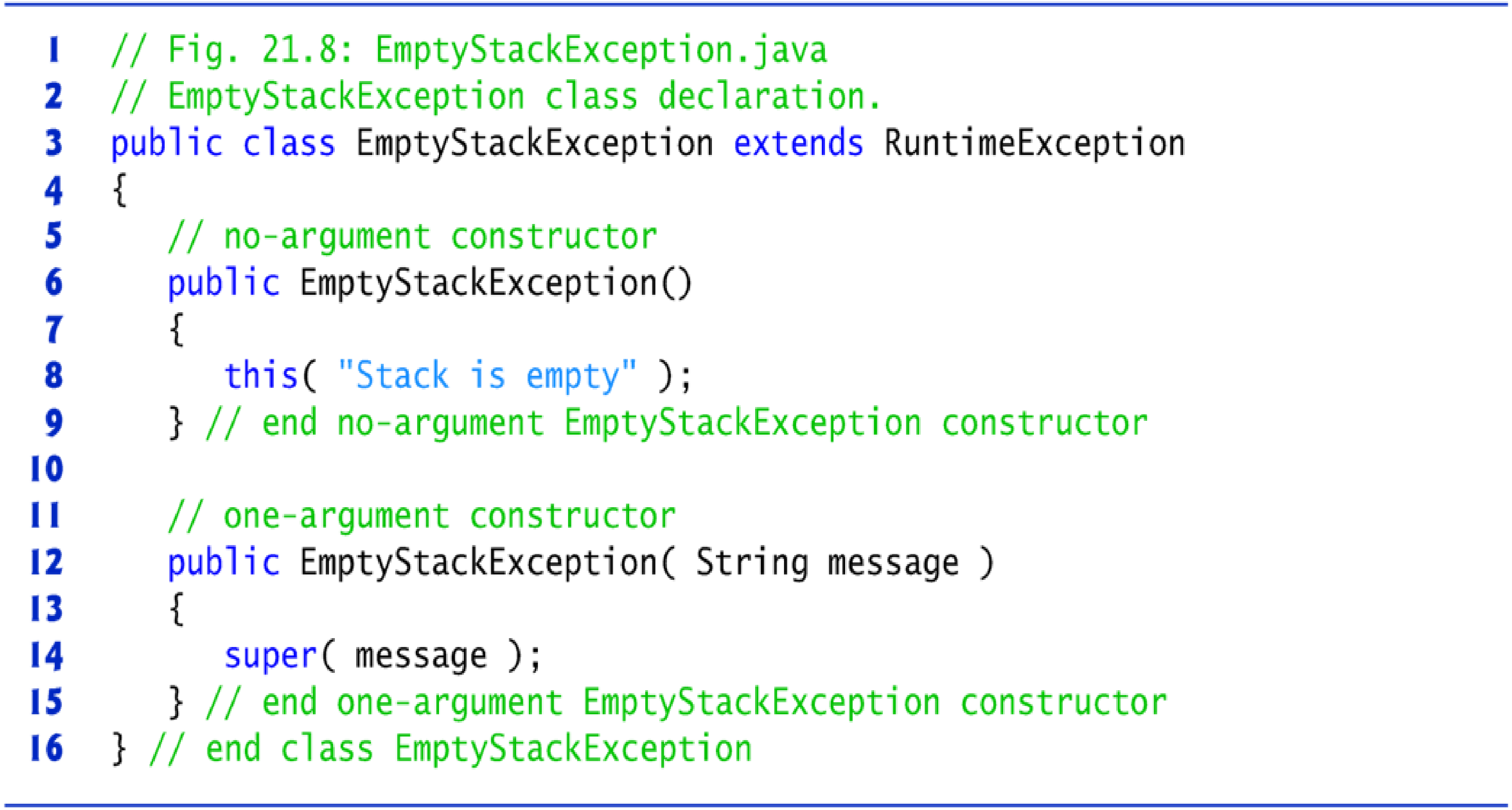
Stack Generic Class



```java
   // push element onto stack
   public void push( T pushValue )
   {
      elements.add( pushValue ); // place pushValue on Stack
   } // end method push

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

      // remove and return top element of Stack
      return elements.remove( elements.size() - 1 );
   } // end method pop
} // end class Stack< T >
```

Fig. 21.7 | Stack generic class declaration. (Part 2 of 2.)

Stack Generic Class



```java
// Fig. 21.8: EmptyStackException.java
// EmptyStackException class declaration.
public class EmptyStackException extends RuntimeException
{
   // no-argument constructor
   public EmptyStackException()
   {
      this( "Stack is empty" );
   } // end no-argument EmptyStackException constructor

   // one-argument constructor
   public EmptyStackException( String message )
   {
      super( message );
   } // end one-argument EmptyStackException constructor
} // end class EmptyStackException
```

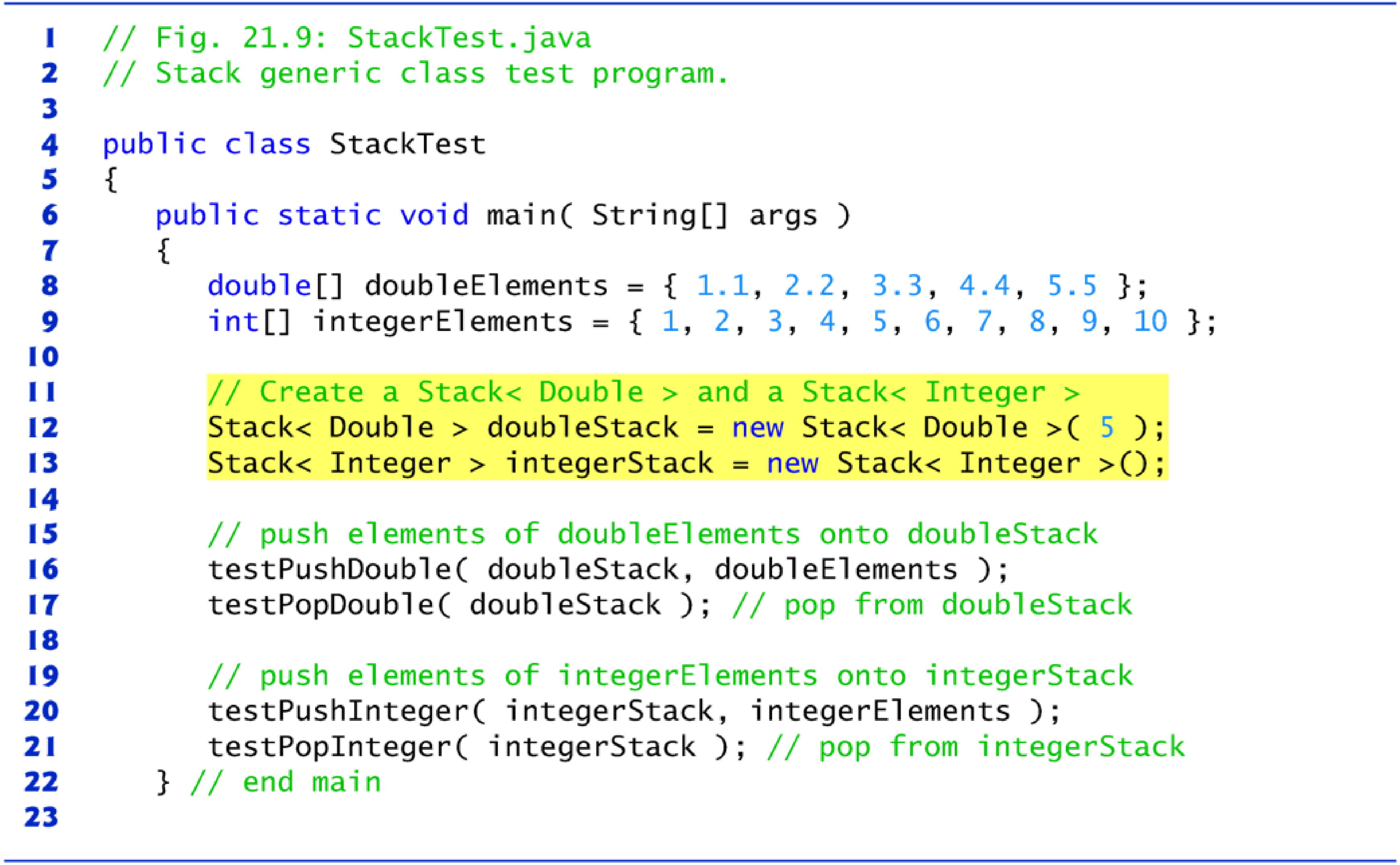
Fig. 21.8 | EmptyStackException class declaration.

Stack Generic Class

```java
// Fig. 21.9: StackTest.java
// Stack generic class test program.

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< Double >( 5 );
      Stack< Integer > integerStack = new Stack< Integer >();

      // 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
   } // end main

```

Fig. 21.9 | Stack generic class test program. (Part 1 of 6.)

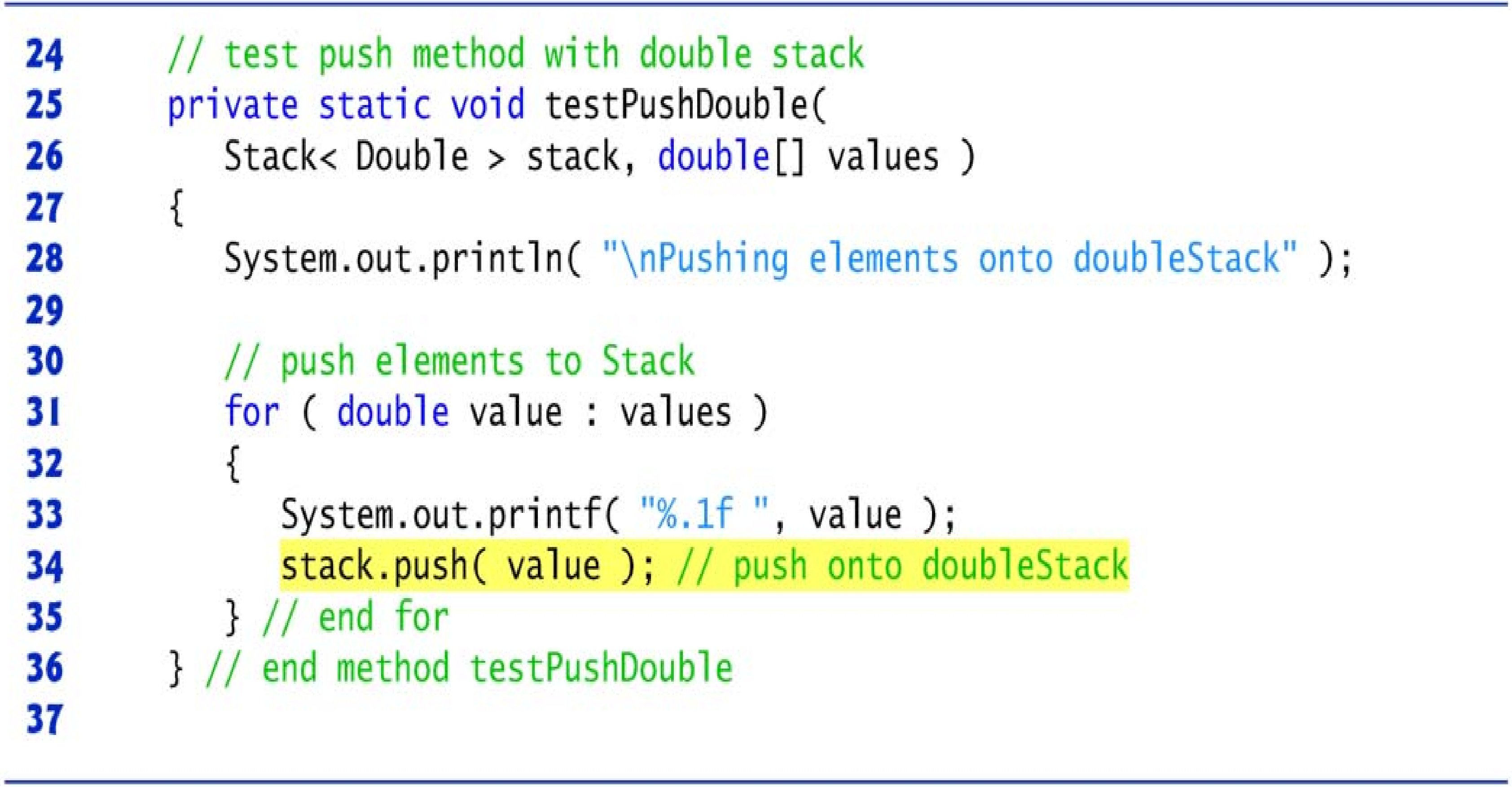
Stack Generic Class



```java
// Fig. 21.9: StackTest.java
// Stack generic class test program.

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< Double >( 5 );
      Stack< Integer > integerStack = new Stack< Integer >();

      // 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
   } // end main

```

Fig. 21.9 | Stack generic class test program. (Part 1 of 6.)

Stack Generic Class



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

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

```

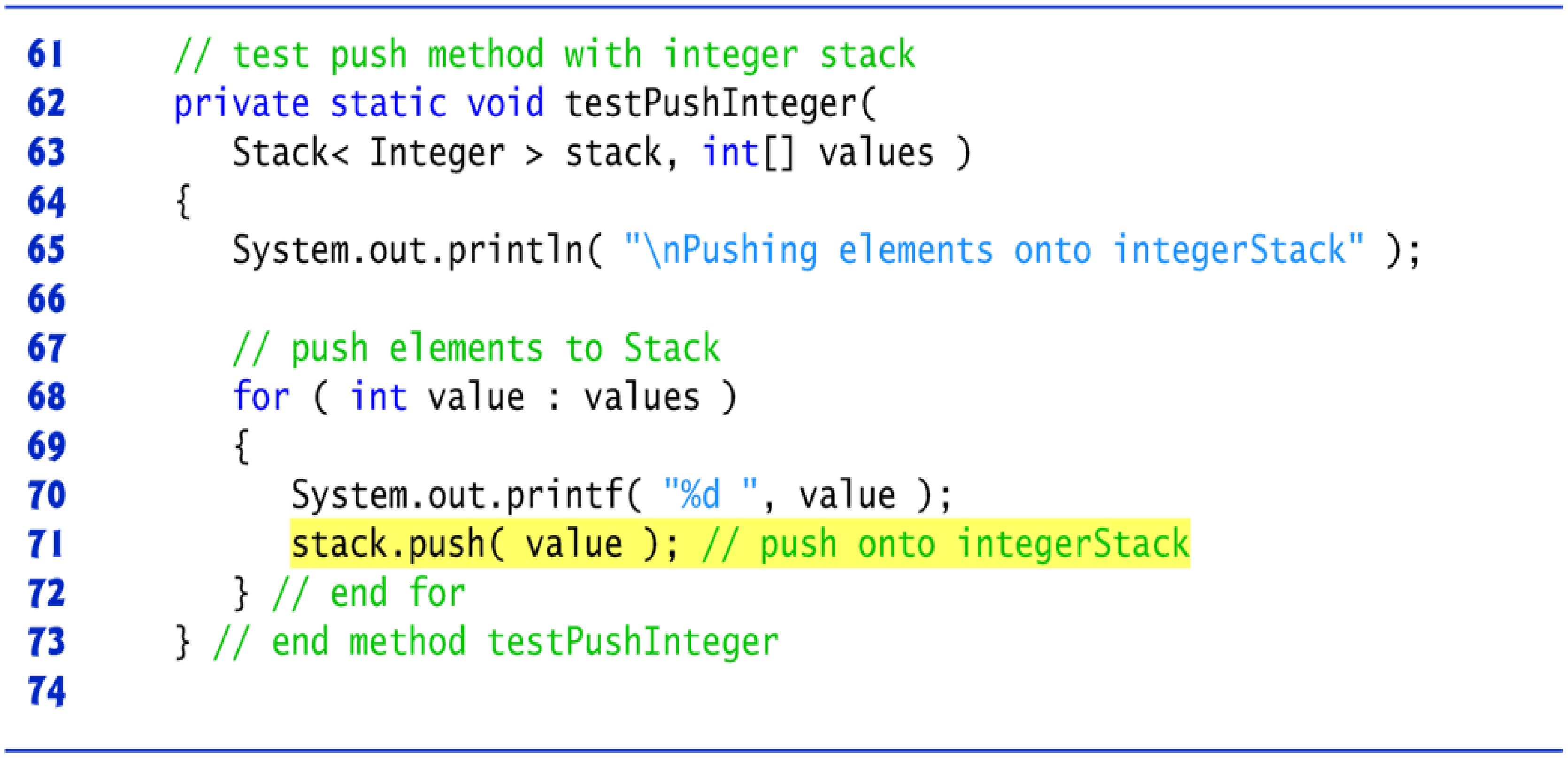
Fig. 21.9 | Stack generic class test program. (Part 2 of 6.)

Stack Generic Class

```java
   // test pop method with double stack
   private static void testPopDouble( Stack< Double > stack )
   {
      // pop elements from stack
      try
      {
         System.out.println( "\nPopping elements from doubleStack" );
         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 );
         } // end while
      } // end try
      catch( EmptyStackException emptyStackException )
      {
         System.err.println();
         emptyStackException.printStackTrace();
      } // end catch EmptyStackException
   } // end method testPopDouble

```

Fig. 21.9 | Stack generic class test program. (Part 3 of 6.)

Stack Generic Class



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

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

```

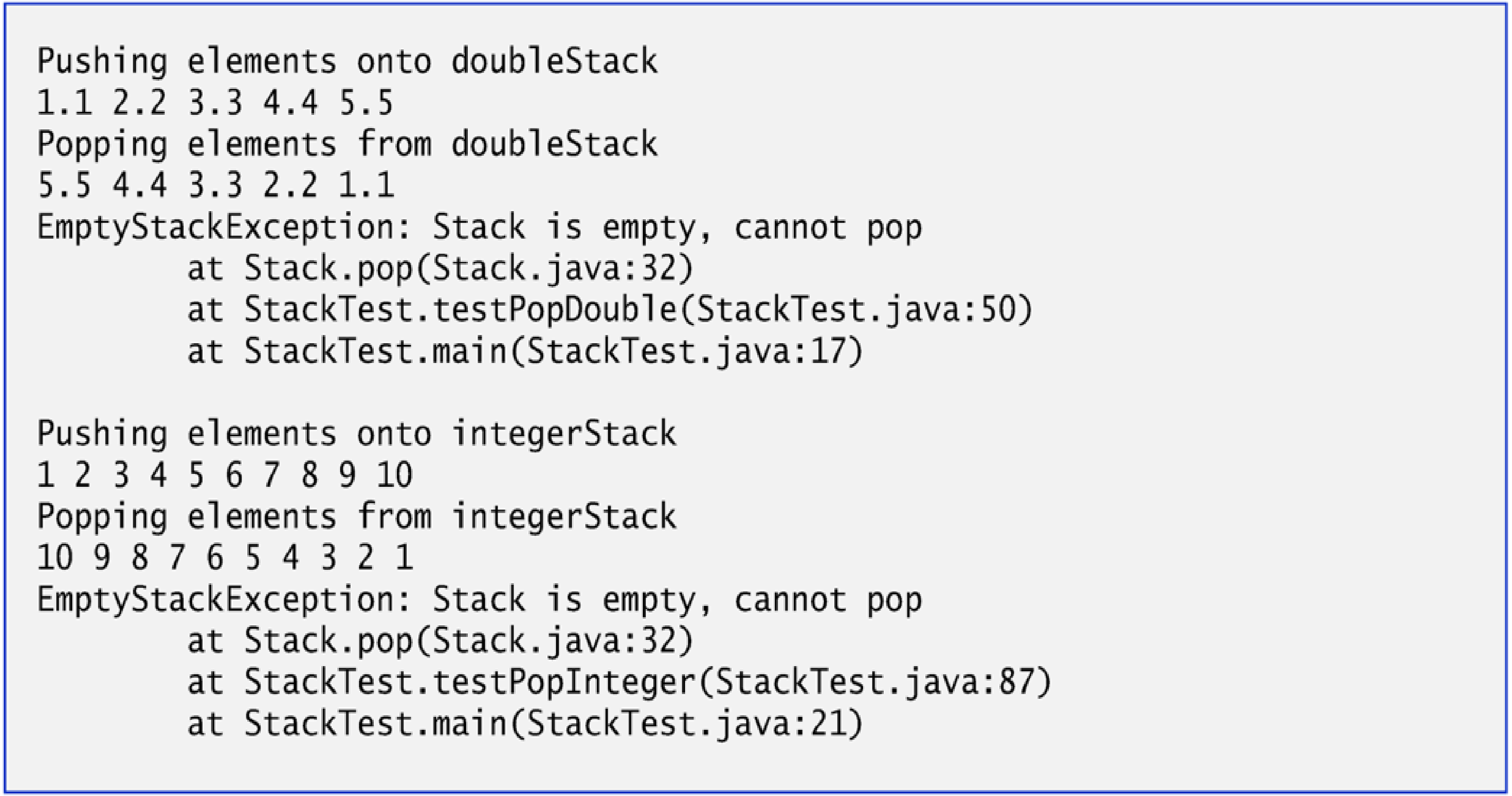
Fig. 21.9 | Stack generic class test program. (Part 4 of 6.)

Stack Generic Class

```java
   // test pop method with integer stack
   private static void testPopInteger( Stack< Integer > stack )
   {
      // pop elements from stack
      try
      {
         System.out.println( "\nPopping elements from integerStack" );
         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 );
         } // end while
      } // end try
      catch( EmptyStackException emptyStackException )
      {
         System.err.println();
         emptyStackException.printStackTrace();
      } // end catch EmptyStackException
   } // end method testPopInteger
} // end class StackTest
```

Fig. 21.9 | Stack generic class test program. (Part 5 of 6.)

Stack Generic Class



```
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
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopDouble(StackTest.java:50)
        at StackTest.main(StackTest.java:17)

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
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopInteger(StackTest.java:87)
        at StackTest.main(StackTest.java:21)
```

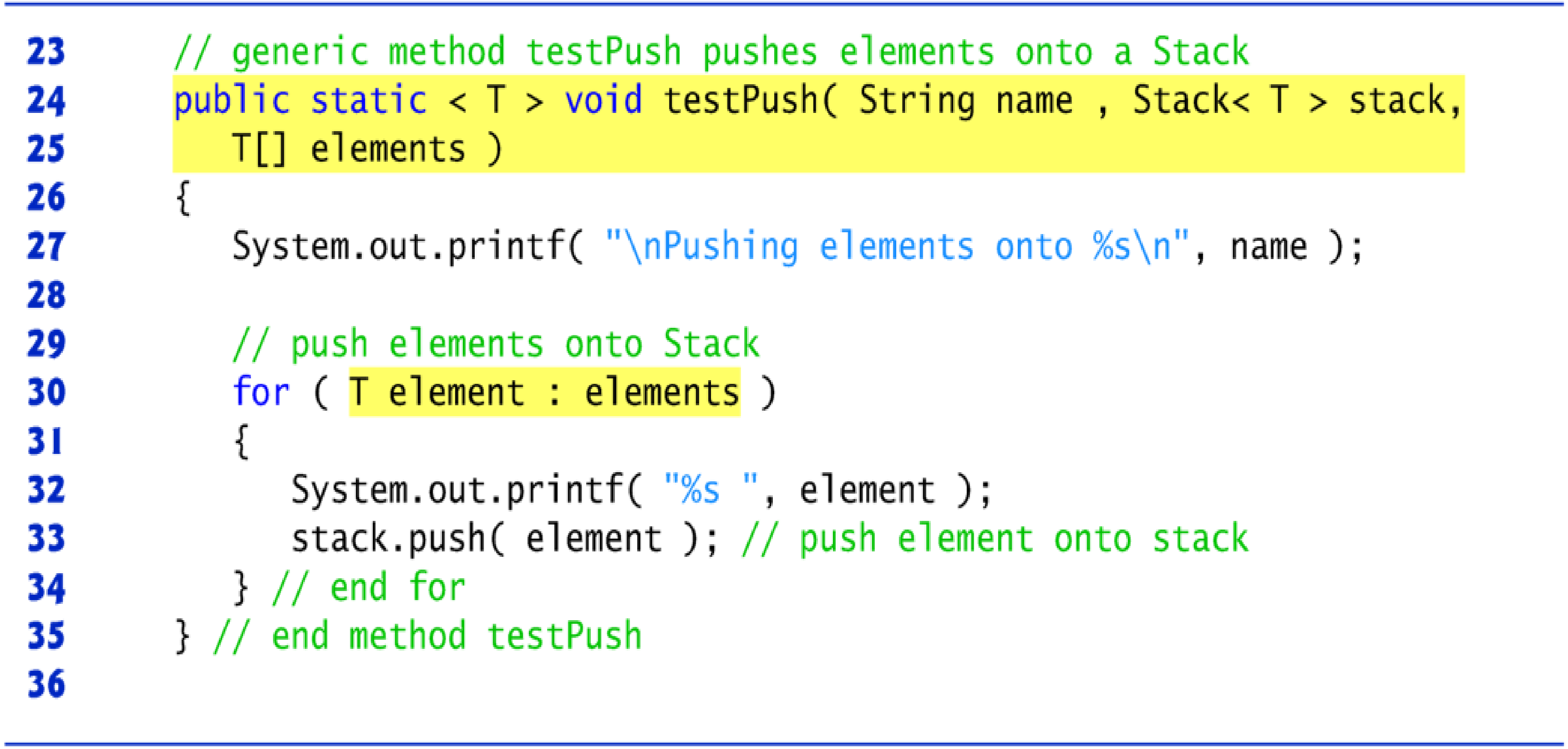
Fig. 21.9 | Stack generic class test program. (Part 6 of 6.)

Stack Generic Class

```java
// Fig. 21.10: StackTest2.java
// Passing generic Stack objects to generic methods.
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< Double >( 5 );
      Stack< Integer > integerStack = new Stack< Integer >();

      // 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
   } // end main

```

Fig. 21.10 | Passing generic Stack objects to generic methods. (Part 1 of 4.)

Stack Generic Class



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

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

```

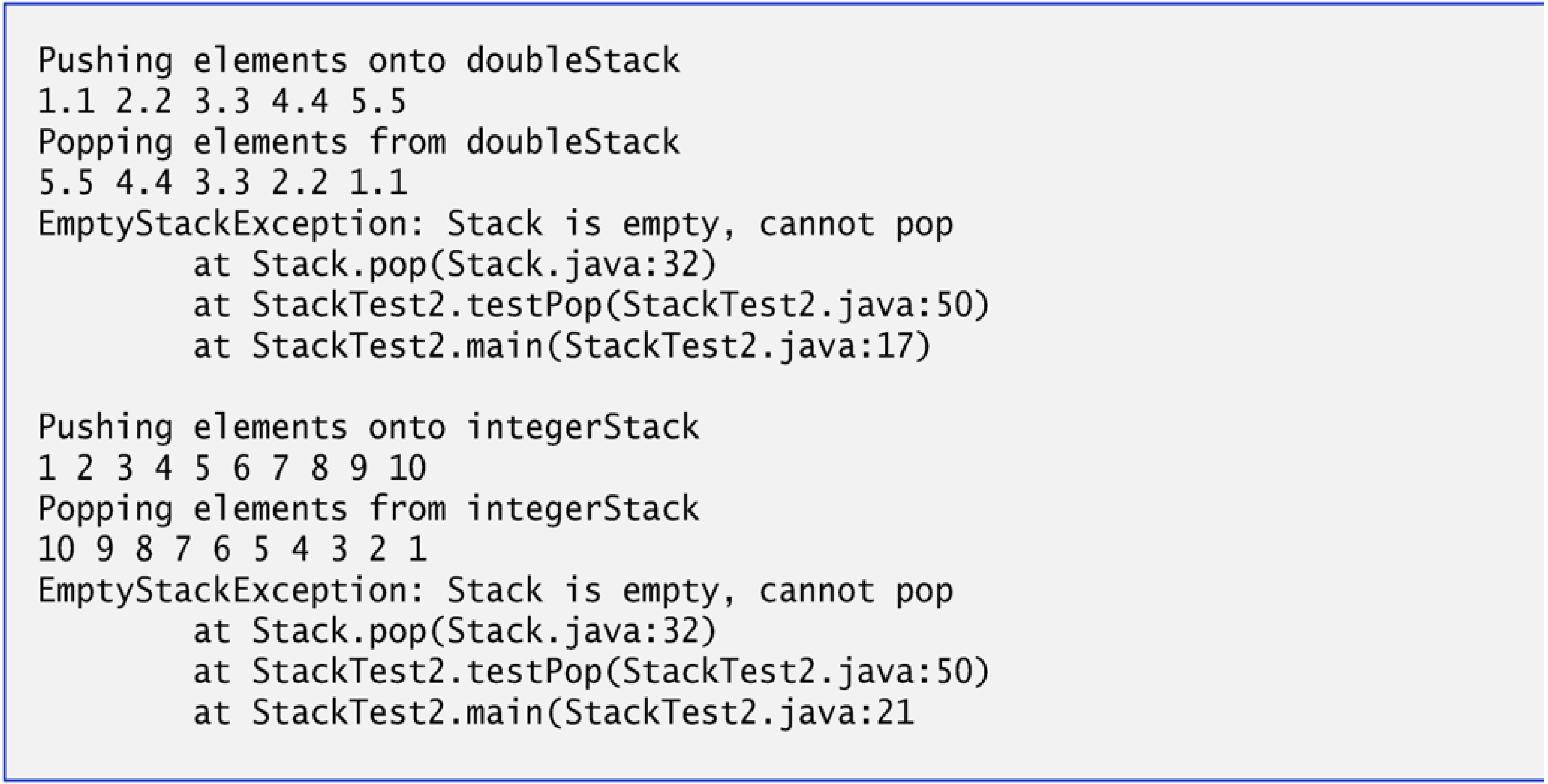
Fig. 21.10 | Passing generic `Stack` objects to generic methods. (Part 2 of 4.)

Stack Generic Class

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

         // remove all elements from Stack
         while ( true )
         {
            popValue = stack.pop();
            System.out.printf( "%s ", popValue );
         } // end while
      } // end try
      catch( EmptyStackException emptyStackException )
      {
         System.out.println();
         emptyStackException.printStackTrace();
      } // end catch EmptyStackException
   } // end method testPop
} // end class StackTest2
```

Fig. 21.10 | Passing generic Stack objects to generic methods. (Part 3 of 4.)

Stack Generic Class



```
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
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:17)

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
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:21
```

Fig. 21.10 | Passing generic `Stack` objects to generic methods. (Part 4 of 4.)