



Chapter 21

Generic Classes and Methods

Java How to Program, 9/e



OBJECTIVES

In this chapter you'll learn:

- To create generic methods that perform identical tasks on arguments of different types.
- To create a generic **Stack** class that can be used to store objects of any class or interface type.
- To understand how to overload generic methods with nongeneric methods or with other generic methods.
- To understand raw types and how they help achieve backward compatibility.
- To use wildcards when precise type information about a parameter is not required in the method body.



21.1 Introduction

21.2 Motivation for Generic Methods

21.3 Generic Methods: Implementation and Compile-Time Translation

21.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type

21.5 Overloading Generic Methods

21.6 Generic Classes

21.7 Raw Types

21.8 Wildcards in Methods That Accept Type Parameters

21.9 Generics and Inheritance: Notes

21.10 Wrap-Up



21.1 Introduction

- ▶ 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.



Software Engineering Observation 21.1

Generic methods and classes are among Java's most powerful capabilities for software reuse with compile-time type safety.



21.2 Motivation for Generic Methods

- ▶ Overloaded methods are often used to perform similar operations on different types of data.
- ▶ Study each `printArray` method.
 - Note that the type array element type appears in each method's header and `for`-statement header.
 - 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. 21.2.



```
1 // Fig. 21.1: OverloadedMethods.java
2 // Printing array elements using overloaded methods.
3 public class OverloadedMethods
4 {
5     public static void main( String[] args )
6     {
7         // create arrays of Integer, Double and Character
8         Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
9         Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
10        Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
11
12        System.out.println( "Array integerArray contains:" );
13        printArray( integerArray ); // pass an Integer array
14        System.out.println( "\nArray doubleArray contains:" );
15        printArray( doubleArray ); // pass a Double array
16        System.out.println( "\nArray characterArray contains:" );
17        printArray( characterArray ); // pass a Character array
18    } // end main
19
```

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



```
20 // method printArray to print Integer array
21 public static void printArray( Integer[] inputArray )
22 {
23     // display array elements
24     for ( Integer element : inputArray )
25         System.out.printf( "%s ", element );
26
27     System.out.println();
28 } // end method printArray
29
30 // method printArray to print Double array
31 public static void printArray( Double[] inputArray )
32 {
33     // display array elements
34     for ( Double element : inputArray )
35         System.out.printf( "%s ", element );
36
37     System.out.println();
38 } // end method printArray
39
```

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



```
40 // method printArray to print Character array
41 public static void printArray( Character[] inputArray )
42 {
43     // display array elements
44     for ( Character element : inputArray )
45         System.out.printf( "%s ", element );
46
47     System.out.println();
48 } // end method printArray
49 } // 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.)



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

Fig. 21.2 | printArray method in which actual type names are replaced by convention with the generic name T.



21.3 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.
- ▶ Line 22 begins method `printArray`'s declaration.



```
1 // Fig. 21.3: GenericMethodTest.java
2 // Printing array elements using generic method printArray.
3
4 public class GenericMethodTest
5 {
6     public static void main( String[] args )
7     {
8         // create arrays of Integer, Double and Character
9         Integer[] intArray = { 1, 2, 3, 4, 5 };
10        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
11        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
12
13        System.out.println( "Array integerArray contains:" );
14        printArray( intArray ); // pass an Integer array
15        System.out.println( "\nArray doubleArray contains:" );
16        printArray( doubleArray ); // pass a Double array
17        System.out.println( "\nArray characterArray contains:" );
18        printArray( charArray ); // pass a Character array
19    } // end main
20}
```

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



```
21 // generic method printArray
22 public static < T > void printArray( T[] inputArray )
23 {
24     // display array elements
25     for ( T element : inputArray )
26         System.out.printf( "%s ", element );
27
28     System.out.println();
29 } // end method printArray
30 } // 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.)



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

- ▶ 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.



Common Programming Error 21.1

When declaring a generic method, failing to place a type-parameter section before the return type of a method is a syntax error—the compiler will not understand the type-parameter names when they're encountered in the method.



Good Programming Practice 21.1

It's recommended that type parameters be specified as individual capital letters. Typically, a type parameter that represents an array element's type (or other collection) is named T.



Common Programming Error 21.2

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



Common Programming Error 21.3

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.

10



21.3 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.



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

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

21.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type

- ▶ Generic method `maximum` determines and returns the largest of its three arguments of the same type.
- ▶ The relational operator `>` cannot be used with reference types, but it's possible to compare two objects of the same class if that class implements the generic interface `Comparable<T>` (package `java.lang`).
 - All the type-wrapper classes for primitive types implement this interface.
- ▶ Like generic classes, `generic interfaces` enable you to specify, with a single interface declaration, a set of related types.

21.4 Additional Compile-Time Translation



Issues: Methods That Use a Type Parameter as the Return Type (cont.)

- ▶ **Comparable<T>** objects have a **compareTo** method.
 - The method must return 0 if the objects are equal, a negative integer if **object1** is less than **object2** or a positive integer if **object1** is greater than **object2**.
- ▶ A benefit of implementing interface **Comparable<T>** is that **Comparable<T>** objects can be used with the sorting and searching methods of class **Collections** (package **java.util**).



```
1 // Fig. 21.5: MaximumTest.java
2 // Generic method maximum returns the largest of three objects.
3
4 public class MaximumTest
5 {
6     public static void main( String[] args )
7     {
8         System.out.printf( "Maximum of %d, %d and %d is %d\n\n",
9             maximum( 3, 4, 5 ) );
10        System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
11            6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
12        System.out.printf( "Maximum of %s, %s and %s is %s\n",
13            "pear", "apple", "orange", maximum( "pear", "apple", "orange" ) );
14    } // end main
15
```

Fig. 21.5 | Generic method `maximum` with an upper bound on its type parameter.
(Part 1 of 2.)



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

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

21.4 Additional Compile-Time Translation



Issues: Methods That Use a Type Parameter as the Return Type (cont.)

- ▶ The type-parameter section specifies that **T** extends **Comparable<T>**—only objects of classes that implement interface **Comparable<T>** can be used with this method.
- ▶ **Comparable** is known as the **upper bound** of the type parameter.
- ▶ By default, **Object** is the upper bound.
- ▶ Type-parameter declarations that bound the parameter always use keyword **extends** regardless of whether the type parameter extends a class or implements an interface.
- ▶ The restriction of using **Comparable<T>** objects is important, because not all objects can be compared.

21.4 Additional Compile-Time Translation



Issues: Methods That Use a Type Parameter as the Return Type (cont.)

- ▶ When the compiler translates generic method `maximum` into Java bytecodes, it uses erasure to replace the type parameters with actual types.
- ▶ All type parameters are replaced with the upper bound of the type parameter, which is specified in the type-parameter section.
- ▶ When the compiler replaces the type-parameter information with the upper-bound type in the method declaration, it also inserts explicit cast operations in front of each method call to ensure that the returned value is of the type expected by the caller.



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

Fig. 21.6 | Generic method `maximum` after erasure is performed by the compiler.



21.5 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.



21.6 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.



```
1 // Fig. 21.7: Stack.java
2 // Stack generic class declaration.
3 import java.util.ArrayList;
4
5 public class Stack< T >
6 {
7     private ArrayList< T > elements; // ArrayList stores stack elements
8
9     // no-argument constructor creates a stack of the default size
10    public Stack()
11    {
12        this( 10 ); // default stack size
13    } // end no-argument Stack constructor
14
15    // constructor creates a stack of the specified number of elements
16    public Stack( int capacity )
17    {
18        int initCapacity = capacity > 0 ? capacity : 10; // validate
19        elements = new ArrayList< T >( initCapacity ); // create ArrayList
20    } // end one-argument Stack constructor
21
```

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



```
22 // push element onto stack
23 public void push( T pushValue )
24 {
25     elements.add( pushValue ); // place pushValue on Stack
26 } // end method push
27
28 // return the top element if not empty; else throw EmptyStackException
29 public T pop()
30 {
31     if ( elements.isEmpty() ) // if stack is empty
32         throw new EmptyStackException( "Stack is empty, cannot pop" );
33
34     // remove and return top element of Stack
35     return elements.remove( elements.size() - 1 );
36 } // end method pop
37 } // end class Stack< T >
```

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



```
1 // Fig. 21.8: EmptyStackException.java
2 // EmptyStackException class declaration.
3 public class EmptyStackException extends RuntimeException
4 {
5     // no-argument constructor
6     public EmptyStackException()
7     {
8         this( "Stack is empty" );
9     } // end no-argument EmptyStackException constructor
10
11    // one-argument constructor
12    public EmptyStackException( String message )
13    {
14        super( message );
15    } // end one-argument EmptyStackException constructor
16 } // end class EmptyStackException
```

Fig. 21.8 | EmptyStackException class declaration.



```
1 // Fig. 21.9: StackTest.java
2 // Stack generic class test program.
3
4 public class StackTest
5 {
6     public static void main( String[] args )
7     {
8         double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
9         int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11     // Create a Stack< Double > and a Stack< Integer >
12     Stack< Double > doubleStack = new Stack< Double >( 5 );
13     Stack< Integer > integerStack = new Stack< Integer >();
14
15     // push elements of doubleElements onto doubleStack
16     testPushDouble( doubleStack, doubleElements );
17     testPopDouble( doubleStack ); // pop from doubleStack
18
19     // push elements of integerElements onto integerStack
20     testPushInteger( integerStack, integerElements );
21     testPopInteger( integerStack ); // pop from integerStack
22 } // end main
23
```

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



```
24 // test push method with double stack
25 private static void testPushDouble(
26     Stack< Double > stack, double[] values )
27 {
28     System.out.println( "\nPushing elements onto doubleStack" );
29
30     // push elements to Stack
31     for ( double value : values )
32     {
33         System.out.printf( "%.1f ", value );
34         stack.push( value ); // push onto doubleStack
35     } // end for
36 } // end method testPushDouble
37
```

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



```
38 // test pop method with double stack
39 private static void testPopDouble( Stack< Double > stack )
40 {
41     // pop elements from stack
42     try
43     {
44         System.out.println( "\nPopping elements from doubleStack" );
45         double popValue; // store element removed from stack
46
47         // remove all elements from Stack
48         while ( true )
49         {
50             popValue = stack.pop(); // pop from doubleStack
51             System.out.printf( "%.1f ", popValue );
52         } // end while
53     } // end try
54     catch( EmptyStackException emptyStackException )
55     {
56         System.err.println();
57         emptyStackException.printStackTrace();
58     } // end catch EmptyStackException
59 } // end method testPopDouble
60
```

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



```
61 // test push method with integer stack
62 private static void testPushInteger(
63     Stack< Integer > stack, int[] values )
64 {
65     System.out.println( "\nPushing elements onto integerStack" );
66
67     // push elements to Stack
68     for ( int value : values )
69     {
70         System.out.printf( "%d ", value );
71         stack.push( value ); // push onto integerStack
72     } // end for
73 } // end method testPushInteger
74
```

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



```
75 // test pop method with integer stack
76 private static void testPopInteger( Stack< Integer > stack )
77 {
78     // pop elements from stack
79     try
80     {
81         System.out.println( "\nPopping elements from integerStack" );
82         int popValue; // store element removed from stack
83
84         // remove all elements from Stack
85         while ( true )
86         {
87             popValue = stack.pop(); // pop from intStack
88             System.out.printf( "%d ", popValue );
89         } // end while
90     } // end try
91     catch( EmptyStackException emptyStackException )
92     {
93         System.err.println();
94         emptyStackException.printStackTrace();
95     } // end catch EmptyStackException
96     } // end method testPopInteger
97 } // end class StackTest
```

Fig. 21.9 | 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
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.)



21.6 Generic Classes (cont.)

- ▶ The code in methods `testPushDouble` and `testPushInteger` from the previous example is almost identical for pushing values onto a `Stack<Double>` or a `Stack<Integer>`, respectively, and the code in methods `testPopDouble` and `testPopInteger` is almost identical for popping values from a `Stack<Double>` or a `Stack<Integer>`, respectively.
- ▶ This presents another opportunity to use generic methods.



```
1 // Fig. 21.10: StackTest2.java
2 // Passing generic Stack objects to generic methods.
3 public class StackTest2
4 {
5     public static void main( String[] args )
6     {
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< Double >( 5 );
12        Stack< Integer > integerStack = new Stack< Integer >();
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    } // end main
22
```

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



```
23 // generic method testPush pushes elements onto a Stack
24 public static < T > void testPush( String name , Stack< T > stack,
25     T[] elements )
26 {
27     System.out.printf( "\nPushing elements onto %s\n", name );
28
29     // push elements onto Stack
30     for ( T element : elements )
31     {
32         System.out.printf( "%s ", element );
33         stack.push( element ); // push element onto stack
34     } // end for
35 } // end method testPush
36
```

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



```
37 // generic method testPop pops elements from a Stack
38 public static < T > void testPop( String name, Stack< T > stack )
39 {
40     // pop elements from stack
41     try
42     {
43         System.out.printf( "\nPopping elements from %s\n", name );
44         T popValue; // store element removed from stack
45
46         // remove all elements from Stack
47         while ( true )
48         {
49             popValue = stack.pop();
50             System.out.printf( "%s ", popValue );
51         } // end while
52     } // end try
53     catch( EmptyStackException emptyStackException )
54     {
55         System.out.println();
56         emptyStackException.printStackTrace();
57     } // end catch EmptyStackException
58 } // end method testPop
59 } // end class StackTest2
```

Fig. 21.10 | 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
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.)



21.8 Raw Types

- ▶ It's also possible to instantiate generic class **Stack** without specifying a type argument, as follows:
 - `// no type-argument specified
Stack objectStack = new Stack(5);`
 - **objectStack** is said to have a **raw type**
 - The compiler implicitly uses type **Object** throughout the generic class for each type argument.
 - The preceding statement creates a **Stack** that can store objects of any type.
 - Important for backward compatibility with prior versions of Java.
 - Raw-type operations are unsafe and could lead to exceptions.



```
1 // Fig. 21.11: RawTypeTest.java
2 // Raw type test program.
3 public class RawTypeTest
4 {
5     public static void main( String[] args )
6     {
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        // Stack of raw types assigned to Stack of raw types variable
11        Stack rawTypeStack1 = new Stack( 5 );
12
13        // Stack< Double > assigned to Stack of raw types variable
14        Stack rawTypeStack2 = new Stack< Double >( 5 );
15
16        // Stack of raw types assigned to Stack< Integer > variable
17        Stack< Integer > integerStack = new Stack( 10 );
18    }
```

Fig. 21.11 | Raw-type test program. (Part 1 of 4.)



```
19     testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
20     testPop( "rawTypeStack1", rawTypeStack1 );
21     testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
22     testPop( "rawTypeStack2", rawTypeStack2 );
23     testPush( "integerStack", integerStack, integerElements );
24     testPop( "integerStack", integerStack );
25 } // end main
26
27 // generic method pushes elements onto stack
28 public static < T > void testPush( String name, Stack< T > stack,
29     T[] elements )
30 {
31     System.out.printf( "\nPushing elements onto %s\n", name );
32
33     // push elements onto Stack
34     for ( T element : elements )
35     {
36         System.out.printf( "%s ", element );
37         stack.push( element ); // push element onto stack
38     } // end for
39 } // end method testPush
40
```

Fig. 21.11 | Raw-type test program. (Part 2 of 4.)



```
41 // generic method testPop pops elements from stack
42 public static < T > void testPop( String name, Stack< T > stack )
43 {
44     // pop elements from stack
45     try
46     {
47         System.out.printf( "\nPopping elements from %s\n", name );
48         T popValue; // store element removed from stack
49
50         // remove elements from Stack
51         while ( true )
52         {
53             popValue = stack.pop(); // pop from stack
54             System.out.printf( "%s ", popValue );
55         } // end while
56     } // end try
57     catch( EmptyStackException emptyStackException )
58     {
59         System.out.println();
60         emptyStackException.printStackTrace();
61     } // end catch EmptyStackException
62 } // end method testPop
63 } // end class RawTypeTest
```

Fig. 21.11 | Raw-type test program. (Part 3 of 4.)



```
Pushing elements onto rawTypeStack1
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack1
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:32)
    at RawTypeTest.testPop(RawTypeTest.java:53)
    at RawTypeTest.main(RawTypeTest.java:20)

Pushing elements onto rawTypeStack2
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack2
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:32)
    at RawTypeTest.testPop(RawTypeTest.java:53)
    at RawTypeTest.main(RawTypeTest.java:22)

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 RawTypeTest.testPop(RawTypeTest.java:53)
    at RawTypeTest.main(RawTypeTest.java:24)
```

Fig. 21.11 | Raw-type test program. (Part 4 of 4.)



21.7 Raw Types (cont.)

- ▶ Figure 21.12 shows the warning messages generated by the compiler when the file `RawTypeTest.java` (Fig. 21.11) is compiled with the `-Xlint:unchecked` option, which provides more information about potentially unsafe operations in code that uses generics.



```
RawTypeTest.java:17: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Integer>
    Stack< Integer > integerStack = new Stack( 10 );
                           ^
RawTypeTest.java:19: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
    testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
                           ^
```

Fig. 21.12 | Warning messages from the compiler. (Part 1 of 3.)



```
RawTypeTest.java:19: warning: [unchecked] unchecked method invocation:  
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to  
(java.lang.String,Stack,java.lang.Double[])  
    testPush( "rawTypeStack1", rawTypeStack1, doubleElements );  
          ^  
RawTypeTest.java:20: warning: [unchecked] unchecked conversion  
found   : Stack  
required: Stack<T>  
    testPop( "rawTypeStack1", rawTypeStack1 );  
          ^  
RawTypeTest.java:20: warning: [unchecked] unchecked method invocation:  
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to  
(java.lang.String,Stack)  
    testPop( "rawTypeStack1", rawTypeStack1 );  
          ^  
RawTypeTest.java:21: warning: [unchecked] unchecked conversion  
found   : Stack  
required: Stack<java.lang.Double>  
    testPush( "rawTypeStack2", rawTypeStack2, doubleElements );  
          ^  
RawTypeTest.java:21: warning: [unchecked] unchecked method invocation:  
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to  
(java.lang.String,Stack,java.lang.Double[])  
    testPush( "rawTypeStack2", rawTypeStack2, doubleElements );  
          ^
```

Fig. 21.12 | Warning messages from the compiler. (Part 2 of 3.)



```
RawTypeTest.java:22: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
    testPop( "rawTypeStack2", rawTypeStack2 );
                           ^
RawTypeTest.java:22: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
    testPop( "rawTypeStack2", rawTypeStack2 );
                           ^
9 warnings
```

Fig. 21.12 | Warning messages from the compiler. (Part 3 of 3.)



21.8 Wildcards in Methods That Accept Type Parameters

- ▶ In this section, we introduce a powerful generics concept known as **wildcards**.
- ▶ Suppose that you'd like to implement a generic method **sum** that totals the numbers in an **ArrayList**.
 - You'd begin by inserting the numbers in the collection.
 - The numbers would be autoboxed as objects of the type-wrapper classes—any `int` value would be autoboxed as an `Integer` object, and any `double` value would be autoboxed as a `Double` object.
 - We'd like to be able to total all the numbers in the **ArrayList** regardless of their type.
 - For this reason, we'll declare the **ArrayList** with the type argument **Number**, which is the superclass of both **Integer** and **Double**.
 - In addition, method **sum** will receive a parameter of type **ArrayList<Number>** and total its elements.



```
1 // Fig. 21.13: TotalNumbers.java
2 // Totaling the numbers in an ArrayList<Number>.
3 import java.util.ArrayList;
4
5 public class TotalNumbers
6 {
7     public static void main( String[] args )
8     {
9         // create, initialize and output ArrayList of Numbers containing
10        // both Integers and Doubles, then display total of the elements
11        Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
12        ArrayList< Number > numberList = new ArrayList< Number >();
13
14        for ( Number element : numbers )
15            numberList.add( element ); // place each number in numberList
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    } // end main
21
```

Fig. 21.13 | Totaling the numbers in an ArrayList<Number>.



```
22 // calculate total of ArrayList elements
23 public static double sum( ArrayList< Number > list )
24 {
25     double total = 0; // initialize total
26
27     // calculate sum
28     for ( Number element : list )
29         total += element.doubleValue();
30
31     return total;
32 } // end method sum
33 } // end class TotalNumbers
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Fig. 21.13 | Totaling the numbers in an `ArrayList<Number>`.



21.8 Wildcards in Methods That Accept Type Parameters (cont.)

- ▶ In method `sum`:
 - The `for` statement assigns each `Number` in the `ArrayList` to variable `element`, then uses `Number` method `doubleValue` to obtain the `Number`'s underlying primitive value as a `double` value.
 - The result is added to `total`.
 - When the loop terminates, the method returns the `total`.



21.8 Wildcards in Methods That Accept Type Parameters (cont.)

- ▶ Given that method `sum` can total the elements of an `ArrayList` of `Numbers`, you might expect that the method would also work for `ArrayLists` that contain elements of only one numeric type, such as `ArrayList<Integer>`.
- ▶ Modified class `TotalNumbers` to create an `ArrayList`- of `Integers` and pass it to method `sum`.
- ▶ When we compile the program, the compiler issues the following error message:
 - `sum(java.util.ArrayList<java.lang.Number>)` in `TotalNumbers` cannot be applied to `(java.util.ArrayList<java.lang.Integer>)`
- ▶ Although `Number` is the superclass of `Integer`, the compiler does not consider the parameterized type `ArrayList<Number>` to be a superclass of `ArrayList<Integer>`.
- ▶ If it were, then every operation we could perform on `ArrayList<Number>` would also work on an `ArrayList<Integer>`.



21.8 Wildcards in Methods That Accept Type Parameters (cont.)

- ▶ To create a more flexible version of the `sum` method that can total the elements of any `ArrayList` containing elements of any subclass of `Number` we use **wildcard-type arguments**.
- ▶ Wildcards enable you to specify method parameters, return values, variables or fields, and so on, that act as supertypes or subtypes of parameterized types.
- ▶ In Fig. 21.14, method `sum`'s parameter is declared in line 50 with the type:
 - `ArrayList< ? extends Number >`
- ▶ A wildcard-type argument is denoted by a question mark (?), which by itself represents an “unknown type.”
 - In this case, the wildcard extends class `Number`, which means that the wildcard has an upper bound of `Number`.
 - Thus, the unknown-type argument must be either `Number` or a subclass of `Number`.



```
1 // Fig. 21.14: WildcardTest.java
2 // Wildcard test program.
3 import java.util.ArrayList;
4
5 public class WildcardTest
6 {
7     public static void main( String[] args )
8     {
9         // create, initialize and output ArrayList of Integers, then
10        // display total of the elements
11        Integer[] integers = { 1, 2, 3, 4, 5 };
12        ArrayList< Integer > integerList = new ArrayList< Integer >();
13
14        // insert elements in integerList
15        for ( Integer element : integers )
16            integerList.add( element );
17
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    }
```

Fig. 21.14 | Generic wildcard test program. (Part I of 4.)



```
22 // create, initialize and output ArrayList of Doubles, then
23 // display total of the elements
24 Double[] doubles = { 1.1, 3.3, 5.5 };
25 ArrayList< Double > doubleList = new ArrayList< Double >();
26
27 // insert elements in doubleList
28 for ( Double element : doubles )
29     doubleList.add( element );
30
31 System.out.printf( "doubleList contains: %s\n", doubleList );
32 System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
33     sum( doubleList ) );
34
35 // create, initialize and output ArrayList of Numbers containing
36 // both Integers and Doubles, then display total of the elements
37 Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
38 ArrayList< Number > numberList = new ArrayList< Number >();
39
```

Fig. 21.14 | Generic wildcard test program. (Part 2 of 4.)



```
40     // insert elements in numberList
41     for ( Number element : numbers )
42         numberList.add( element );
43
44     System.out.printf( "numberList contains: %s\n", numberList );
45     System.out.printf( "Total of the elements in numberList: %.1f\n",
46         sum( numberList ) );
47 } // end main
48
49 // total the elements; using a wildcard in the ArrayList parameter
50 public static double sum( ArrayList< ? extends Number > list )
51 {
52     double total = 0; // initialize total
53
54     // calculate sum
55     for ( Number element : list )
56         total += element.doubleValue();
57
58     return total;
59 } // end method sum
60 } // end class WildcardTest
```

Fig. 21.14 | Generic wildcard test program. (Part 3 of 4.)



```
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. 21.14 | Generic wildcard test program. (Part 4 of 4.)



21.8 Wildcards in Methods That Accept Type Parameters (cont.)

- ▶ Because the wildcard (?) in the method's header does not specify a type-parameter name, you cannot use it as a type name throughout the method's body (i.e., you cannot replace **Number** with ? in line 55).
- ▶ You could, however, declare method **sum** as follows:
 - `public static <T extends Number> double sum(ArrayList< T > list)`
 - allows the method to receive an **ArrayList** that contains elements of any **Number** subclass.
 - You could then use the type parameter **T** throughout the method body.
- ▶ If the wildcard is specified without an upper bound, then only the methods of type **Object** can be invoked on values of the wildcard type.
- ▶ Also, methods that use wildcards in their parameter's type arguments cannot be used to add elements to a collection referenced by the parameter.



Common Programming Error 21.4

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.