

18

Generics



Every man of genius sees the world at a different angle from his fellows.

— Havelock Ellis

...our special individuality, as distinguished from our generic humanity.

— Oliver Wendell Holmes, Sr.

Born under one law, to another bound.

— Lord Brooke

You deal in the raw material of opinion, and, if my convictions have any validity, opinion ultimately governs the world.

— Woodrow Wilson



OBJECTIVES

In this chapter you will 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 non-generic methods or with other generic methods.
- To understand raw types and how they help achieve backwards compatibility.
- To use wildcards when precise type information about a parameter is not required in the method body.
- The relationship between generics and inheritance.



- 18.1 Introduction**
- 18.2 Motivation for Generic Methods**
- 18.3 Generic Methods: Implementation and Compile-Time Translation**
- 18.4 Additional Compile-Time Translation Issues:**
- 18.5 Overloading Generic Methods**
- 18.6 Generic Classes**
- 18.7 Raw Types**
- 18.8 Wildcards in Methods That Accept Type Parameters**
- 18.9 Generics and Inheritance: Notes**
- 18.10 Wrap-Up**
- 18.11 Internet and Web Resources**



18.1 Introduction

- **Generics**

- **New feature of J2SE 5.0**
- **Provide compile-time type safety**
 - **Catch invalid types at compile time**
- **Generic methods**
 - **A single method declaration**
 - **A set of related methods**
- **Generic classes**
 - **A single class declaration**
 - **A set of related classes**



Software Engineering Observation 18.1

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



18.2 Motivation for Generic Methods

- **Overloaded methods**

- Perform similar operations on different types of data
- Overloaded `printArray` methods
 - Integer array
 - Double array
 - Character array
- Only reference types can be used with generic methods and classes



```
1 // Fig. 18.1: OverloadedMethods.java
2 // Using overloaded methods to print array of different types.
3
4 public class OverloadedMethods
5 {
6     // method printArray to print Integer array
7     public static void printArray( Integer[] inputArray )
8     {
9         // display array elements
10        for ( Integer element : inputArray )
11            System.out.printf( "%s ", element );
12
13        System.out.println();
14    } // end method printArray
15
16    // method printArray to print Double array
17    public static void printArray( Double[] inputArray )
18    {
19        // display array elements
20        for ( Double element : inputArray )
21            System.out.printf( "%s ", element );
22
23        System.out.println();
24    } // end method printArray
25
```

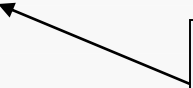
Method printArray accepts
an array of Integer objects

Method printArray accepts
an array of Double objects




```
26 // method printArray to print Character array
27 public static void printArray( Character[] inputArray )
28 {
29     // display array elements
30     for ( Character element : inputArray )
31         System.out.printf( "%s ", element );
32
33     System.out.println();
34 } // end method printArray
35
36 public static void main( String args[] )
37 {
38     // create arrays of Integer, Double and Character
39     Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
40     Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
41     Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
42
```

Method printArray accepts
an array of Character objects



```
43 System.out.println( "Array integerArray contains:" );
44 printArray( integerArray ); // pass an Integer array
45 System.out.println( "\nArray doubleArray contains:" );
46 printArray( doubleArray ); // pass a Double array
47 System.out.println( "\nArray characterArray contains:" );
48 printArray( characterArray ); // pass a Character array
49 } // end main
50 } // end class OverloadedMethods
```

At compile time, the compiler determines argument **integerArray**'s type (i.e., **Integer[]**), attempts to locate a method named **printArray** that specifies a single **Integer[]** parameter (lines 7-14)

At compile time, the compiler determines argument **doubleArray**'s type (i.e., **Double[]**), attempts to locate a method named **printArray** that specifies a single **Double[]** parameter (lines 17-24)

At compile time, the compiler determines argument **characterArray**'s type (i.e., **Character[]**), attempts to locate a method named **printArray** that specifies a single **Character[]** parameter (lines 27-34)

```
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
```



18.2 Motivation for Generic Methods (Cont.)

- **Study each `printArray` method**
 - **Array element type appears in two location**
 - **Method header**
 - **for statement header**
- **Combine three `printArray` methods into one**
 - **Replace the element types with a generic name `E`**
 - **Declare one `printArray` method**
 - **Display the string representation of the elements of any array**



```
1 public static void printArray( E[] inputArray )
2 {
3     // display array elements
4     for ( E element : inputArray )
5         system.out.print
6
7     system.out.println
8 } // end method printArray
```

Replace the element type with a single generic type E

Replace the element type with a single generic type E

Fig. 18.2 | printArray method in which actual type names are replaced by convention with the generic name E.



18.3 Generic Methods: Implementation and Compile-Time Translation

- **Reimplement Fig. 18.1 using a generic method**
 - Method calls are identical
 - Outputs are identical
- **Generic method declaration**
 - Type parameter section
 - Delimited by angle brackets (< and >)
 - Precede the method's return type
 - Contain one or more type parameters
 - Also called formal type parameters



18.3 Generic Methods: Implementation and Compile-Time Translation

- **Type parameter**
 - Also known as type variable
 - An identifier that specifies a generic type name
 - Used to declare return type, parameter types and local variable types
 - Act as placeholders for the types of the argument passed to the generic method
 - Actual type arguments
 - Can be declared only once but can appear more than once
- ```
public static < E > void printTwoArrays(
 E[] array1, E[] array2)
```



# Common Programming Error 18.1

---

**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 name when it is encountered in the method.**



```
1 // Fig. 18.3: GenericMethodTest.java
2 // Using generic methods to print array of different types.
3
4 public class GenericMethodTest
5 {
6 // generic method printArray
7 public static < E > void printArray(E[] inputArray)
8 {
9 // display array elements
10 for (E element : inputArray)
11 System.out.printf("%s ", element);
12
13 System.out.println();
14 } // end method printArray
15
16 public static void main(String args[])
17 {
18 // create arrays of Integer, Double and Character
19 Integer[] intArray = { 1, 2, 3, 4, 5 };
20 Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
21 Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
22 }
```

Use the type parameter to declare method `printArray`'s parameter type

Type parameter section delimited by angle brackets (< and > )

Use the type parameter to declare method `printArray`'s local variable type





```
23 System.out.println("Array integerArray contains:");
24 printArray(integerArray); // pass an Integer array
25 System.out.println("\nArray doubleArray contains:");
26 printArray(doubleArray); // pass a Double array
27 System.out.println("\nArray characterArray contains:");
28 printArray(characterArray); // pass a Character array
29 } // end main
30 } // end class GenericMethodTest
```

Invoke generic method `printArray` with an **Integer** array

Invoke generic method `printArray` with a **Double** array

Invoke generic method `printArray` with a **Character** array

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



# Good Programming Practice 18.1

---

**It is recommended that type parameters be specified as individual capital letters. Typically, a type parameter that represents the type of an element in an array (or other collection) is named E for “element.”**



# Common Programming Error 18.2

---

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



# Common Programming Error 18.3

---

**If the compiler does not find a method declaration that matches a method call exactly, but does find two or more generic methods that can satisfy the method call, a compilation error occurs.**



## 18.3 Generic Methods: Implementation and Compile-Time Translation (Cont.)

- **Compile-time translation**
  - **Erasure**
    - Remove type parameter section
    - Replace type parameters with actual types
    - Default type is `Object`



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

Remove type parameter section and replace type parameter with actual type **Object**

Replace type parameter with actual type **Object**

**Fig. 18.4 |** Generic method `printArray` after erasure is performed by the compiler.



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

- **Application of Fig. 18.5**
  - Generic method
  - Use Type parameters in the return type and parameter list
- **Generic interface**
  - Specify, with a single interface declaration, a set of related types
  - E.g., `Comparable<T>`
    - Method `integer1.compareTo( integer2 )`
      - Compare two objects of the same class
      - Return 0 if two objects are equal
      - Return -1 if `integer1` is less than `integer2`
      - Return 1 if `integer1` is greater than `integer2`

```

1 // Fig. 18.5: MaximumTest.java
2 // Generic method maximum returns the largest of three objects.
3
4 public class MaximumTest
5 {
6 // determines the largest of three Comparable objects
7 public static < T extends Comparable< T > > T maximum(T x, T y, T z)
8 {
9 T max = x; // assume x is initially the largest
10
11 if (y.compareTo(max) > 0)
12 max = y; // y is the largest so far
13
14 if (z.compareTo(max) > 0)
15 max = z; // z is the largest
16
17 return max; // returns the largest object
18 } // end method maximum
19

```

Type parameter

Type parameter is used in the return type of method `maximum`

`Comparable` can be used with this method

Invokes method `compareTo` method `Comparable` to compare `y` and `max`

Invokes method `compareTo` method `Comparable` to compare `z` and `max`

Assign `x` to local variable `max`





```
20 public static void main(String args[])
21 {
22 System.out.printf("Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
23 maximum(3, 4, 5));
24 System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
25 6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
26 System.out.printf("Maximum of %s, %s and %s is %s\n\n",
27 "apple", "orange", maximum("pear", "apple", "orange"));
28 } // end main
29 } // end class MaximumTest
```

Invoke generic method  
maximum with three integers

Invoke generic method  
maximum with three doubles

Invoke generic method  
maximum with three strings

```
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
```



## 18.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type (Cont.)

- **Upper bound of type parameter**
    - **Default is Object**
    - Always use keyword **extends**
      - E.g., `T extends Comparable< T >`
    - When compiler translates generic method to Java bytecode
      - Replaces type parameter with its upper bound
      - Insert explicit cast operation
- e.g., line 23 of Fig. 18.5 I preceded by an **Integer** cast
- ```
(Integer) maximum( 3, 4, 5 )
```



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

Erasure replaces type parameter T with its upper bound Comparable

Erasure replaces type parameter T with its upper bound Comparable



18.5 Overloading Generic Method

- **Generic method may be overloaded**
 - **By another generic method**
 - Same method name but different method parameters
 - **By non-generic methods**
 - Same method name and number of parameters
- **When compiler encounters a method call**
 - **Search for most precise matching method first**
 - Exact method name and argument types
 - **Then search for inexact but applicable matching method**



18.6 Generic Classes

- **Generic classes**

- Use a simple, concise notation to indicate the actual type(s)
- At compilation time, Java compiler
 - ensures the type safety
 - uses the erasure technique to enable client code to interact with the generic class

- **Parameterized classes**

- Also called parameterized types
- E.g., `Stack< Double >`



18.6 Generic Classes (Cont.)

- **Generic class declaration**
 - Looks like a non-generic class declaration
 - Except class name is followed by a type parameter section
- **The `-Xlint:unchecked` option**
 - Compiler cannot 100% ensure type safety



```

1 // Fig. 18.7: Stack.java
2 // Generic class Stack.
3
4 public class Stack< E >
5 {
6     private final int size; // number
7     private int top; // location of top element
8     private E[] elements; // array that stores stack elements
9
10    // no-argument constructor creates stack of size 10
11    public Stack()
12    {
13        this( 10 ); // default stack size
14    } // end no-argument Stack constructor
15
16    // constructor creates a stack of the specified number of elements
17    public Stack( int s )
18    {
19        size = s > 0 ? s : 10; // set size of stack
20        top = -1; // stack initially empty
21
22        elements = ( E[] ) new Object[ size ]; // create array
23    } // end Stack constructor
24

```

Generic class declaration, class name is followed by a type parameter section

Declare `elements` as an array that stores objects of type `E`

Create an array of type `E`. The generic mechanism does not allow type parameter in array-creation expressions because the type parameter is not available at runtime



```

25 // push element onto stack; if successful, return true;
26 // otherwise, throw FullStackException
27 public void push( E pushValue )
28 {
29     if ( top == size - 1 ) // if stack is
30         throw new FullStackException( String
31             "Stack is full, cannot push %s", pushValue ) );
32
33     elements[ ++top ] = pushValue; // place pushValue on stack
34 } // end method push
35
36 // return the top element if not empty; else throw EmptyStackException
37 public E pop()
38 {
39     if ( top == -1 ) // if
40         throw new EmptyStackException( "Stack is empty, cannot pop" );
41
42     return elements[ top-- ]; // remove and return top element of stack
43 } // end method pop
44 } // end class Stack< E >

```

Method **push** pushes
element of type **E** onto stack

Method **pop** returns the top
element, which is of type **E**




```
1 // Fig. 18.8: FullStackException.java
2 // Indicates a stack is full.
3 public class FullStackException extends RuntimeException
4 {
5     // no-argument constructor
6     public FullStackException()
7     {
8         this( "Stack is full" );
9     } // end no-argument FullStackException constructor
10
11     // one-argument constructor
12     public FullStackException( String exception )
13     {
14         super( exception );
15     } // end one-argument FullStackException constructor
16 } // end class FullStackException
```



```
1 // Fig. 18.9: EmptyStackException.java
2 // Indicates a stack is full.
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 exception )
13     {
14         super( exception );
15     } // end one-argument EmptyStackException constructor
16 } // end class EmptyStackException
```



18.6 Generic Classes (Cont.)

- **Generic class at compilation time**
 - Compiler performs erasure on class's type parameters
 - Compiler replaces type parameters with their upper bound
- **Generic class test program at compilation time**
 - Compiler performs type checking
 - Compiler inserts cast operations as necessary



```
1 // Fig. 18.10: StackTest.java
2 // Stack generic class test program.
3
4 public class StackTest
5 {
6     private double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
8
9     private Stack< Double > doubleStack; // stack stores Double objects
10    private Stack< Integer > integerStack; // stack of Integer objects
11
12    // test stack objects
13    public void testStacks()
14    {
15        doubleStack = new Stack< Double >( 5 );
16        integerStack = new Stack< Integer >( 10 ); // stack of Integer objects
17
18        testPushDouble(); // push double onto doubleStack
19        testPopDouble(); // pop from doubleStack
20        testPushInteger(); // push int onto intStack
21        testPopInteger(); // pop from intStack
22    } // end method testStacks
23
```

Generic class Stack's type argument is Double

Generic class Stack's type argument is Integer

Instantiate object doubleStack of size 5 and integerStack of size 10



```
24 // test push method with double stack
25 public void testPushDouble()
26 {
27     // push elements onto stack
28     try
29     {
30         System.out.println( "\nPushing elements onto doubleStack" );
31
32         // push elements to stack
33         for ( double element : doubleElements )
34         {
35             System.out.printf( "%.1f ", element );
36             doubleStack.push( element ); // push onto doubleStack
37         } // end for
38     } // end try
39     catch ( FullStackException fullStackException )
40     {
41         System.err.println();
42         fullStackException.printStackTrace();
43     } // end catch FullStackException
44 } // end method testPushDouble
45
```

Invoke Stack's method push to place a double value onto doubleStack



```
46 // test pop method with double stack
47 public void testPopDouble()
48 {
49     // pop elements from stack
50     try
51     {
52         System.out.println( "\nPopping elements from doubleStack" );
53         double popValue; // store element removed from stack
54
55         // remove all elements from stack
56         while ( true )
57         {
58             popValue = doubleStack.pop(); // pop from doubleStack
59             System.out.printf( "%.1f ", popValue );
60         } // end while
61     } // end try
62     catch( EmptyStackException emptyStackException )
63     {
64         System.err.println();
65         emptyStackException.printStackTrace();
66     } // end catch EmptyStackException
67 } // end method testPopDouble
68
```

Auto-unboxing occurs when the value returned by pop (Double) is assigned to a double primitive variable



```
69 // test push method with integer stack
70 public void testPushInteger()
71 {
72     // push elements to stack
73     try
74     {
75         System.out.println( "\nPushing elements onto intStack" );
76
77         // push elements to stack
78         for ( int element : integerElements )
79         {
80             System.out.printf( "%d ", element );
81             integerStack.push( element ); // push onto integerStack
82         } // end for
83     } // end try
84     catch ( FullStackException fullStackException )
85     {
86         System.err.println();
87         fullStackException.printStackTrace();
88     } // end catch FullStackException
89 } // end method testPushInteger
90
```

Invoke Stack's method push to place an int value onto integerStack



```
91 // test pop method with integer stack
92 public void testPopInteger()
93 {
94     // pop elements from stack
95     try
96     {
97         System.out.println( "\nPopping elements from intStack" );
98         int popValue; // store element removed from stack
99
100        // remove all elements from Stack
101        while ( true )
102        {
103            popValue = integerStack.pop(); // pop from intStack
104            System.out.printf( "%d ", popValue );
105        } // end while
106    } // end try
107    catch( EmptyStackException emptyStackException )
108    {
109        System.err.println();
110        emptyStackException.printStackTrace();
111    } // end catch EmptyStackException
112 } // end method testPopInteger
113
114 public static void main( String args[] )
115 {
116     StackTest application = new StackTest();
117     application.testStacks();
118 } // end main
119 } // end class StackTest
```

Auto-unboxing occurs when the value returned by `pop (Integer)` is assigned to an `int` primitive variable



Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5 6.6

FullStackException: Stack is full, cannot push 6.6

```
at Stack.push(Stack.java:30)
at StackTest.testPushDouble(StackTest.java:36)
at StackTest.testStacks(StackTest.java:18)
at StackTest.main(StackTest.java:117)
```

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:40)
at StackTest.testPopDouble(StackTest.java:58)
at StackTest.testStacks(StackTest.java:19)
at StackTest.main(StackTest.java:117)
```

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

FullStackException: Stack is full, cannot push 11

```
at Stack.push(Stack.java:30)
at StackTest.testPushInteger(StackTest.java:81)
at StackTest.testStacks(StackTest.java:20)
at StackTest.main(StackTest.java:117)
```

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:40)
at StackTest.testPopInteger(StackTest.java:103)
at StackTest.testStacks(StackTest.java:21)
at StackTest.main(StackTest.java:117)
```



18.6 Generic Classes (Cont.)

- **Creating generic methods to test class `Stack< E >`**
 - **Method `testPush`**
 - Perform same tasks as `testPushDouble` and `testPushInteger`
 - **Method `testPop`**
 - Perform same tasks as `testPopDouble` and `testPopInteger`



```
1 // Fig. 18.11: StackTest2.java
2 // Stack generic class test program.
3
4 public class StackTest2
5 {
6     private Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private Integer[] integerElements =
8         { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
9
10    private Stack< Double > doubleStack; // stack stores Double objects
11    private Stack< Integer > integerStack; // stack stores Integer objects
12
13    // test stack objects
14    public void testStacks()
15    {
16        doubleStack = new Stack< Double >( 5 ); // stack of Doubles
17        integerStack = new Stack< Integer >( 10 ); // stack of Integers
18
19        testPush( "doubleStack", doubleStack, doubleElements );
20        testPop( "doubleStack", doubleStack );
21        testPush( "integerStack", integerStack, integerElements );
22        testPop( "integerStack", integerStack );
23    } // end method testStacks
24
```

Invoke generic methods **testPush** and **testPop** to push elements onto stack and pop elements from stack



```

25 // generic method testPush pushes elements onto a Stack
26 public < T > void testPush( String name, Stack< T > stack,
27     T[] elements )
28 {
29     // push elements onto stack
30     try
31     {
32         System.out.printf( "\nPushing elements onto %s\n", name );
33
34         // push elements onto Stack
35         for ( T element : elements )
36         {
37             System.out.printf( "%s ", element );
38             stack.push( element ); // push element onto stack
39         }
40     } // end try
41     catch ( FullStackException fullStackException )
42     {
43         System.out.println();
44         fullStackException.printStackTrace();
45     } // end catch FullStackException
46 } // end method testPush
47

```

Generic method `testPush` replaces `testPushDouble` and `testPushInteger`

Replace element type `Double/Integer` with type parameter `T`



```

48 // generic method testPop pops elements from a Stack
49 public < T > void testPop( String name, Stack< T > stack )

```

Generic method testPop replaces
testPopDouble and testPopInteger

```

50 {
51     // pop elements from stack
52     try
53     {
54         System.out.printf( "\nPopping elements from %s\n", name );
55         T popValue; // store element removed from stack

```

Replace element type Double/Integer
with type parameter T

```

56
57     // remove elements from Stack
58     while ( true )
59     {
60         popValue = stack.pop(); // pop from stack
61         System.out.printf( "%s ", popValue );
62     } // end while
63 } // end try
64 catch( EmptyStackException emptyStackException )
65 {
66     System.out.println();
67     emptyStackException.printStackTrace();
68 } // end catch EmptyStackException
69 } // end method testPop

```

```

70
71 public static void main( String args[] )
72 {
73     StackTest2 application = new StackTest2();
74     application.testStacks();
75 } // end main
76 } // end class StackTest2

```



Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5 6.6

```
FullStackException: Stack is full, cannot push 6.6
    at Stack.push(Stack.java:30)
    at StackTest2.testPush(StackTest2.java:38)
    at StackTest2.testStacks(StackTest2.java:19)
    at StackTest2.main(StackTest2.java:74)
```

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:40)
    at StackTest2.testPop(StackTest2.java:60)
    at StackTest2.testStacks(StackTest2.java:20)
    at StackTest2.main(StackTest2.java:74)
```

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

```
FullStackException: Stack is full, cannot push 11
    at Stack.push(Stack.java:30)
    at StackTest2.testPush(StackTest2.java:38)
    at StackTest2.testStacks(StackTest2.java:21)
    at StackTest2.main(StackTest2.java:74)
```

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:40)
    at StackTest2.testPop(StackTest2.java:60)
    at StackTest2.testStacks(StackTest2.java:22)
    at StackTest2.main(StackTest2.java:74)
```



18.7 Raw Types

- **Raw type**

- **Enables** to instantiate generic class without specifying a type argument

e.g., `Stack objectStack = new Stack(5);`

`objectStack` is said to have a raw type

- Important for backwards compatibility with prior versions
- A raw type `Stack` variable can be assigned a `Stack` that specifies a type argument
- A `Stack` variable that specifies a type argument can be assigned a raw type `Stack`
 - Permitted but unsafe
 - Use the `-Xlint:unchecked` option to compile



```

1 // Fig. 18.12: RawTypeTest.java
2 // Raw type test program.
3
4 public class RawTypeTest
5 {
6     private Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private Integer[] integerElements =
8         { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
9
10    // method to test Stacks with raw types
11    public void testStacks()
12    {
13        // Stack of raw types assigned to stack of raw types variable
14        Stack rawTypeStack1 = new Stack( 5 );
15
16        // Stack< Double > assigned to stack of raw types variable
17        Stack rawTypeStack2 = new Stack< Double >( 5 );
18
19        // Stack of raw types assigned to Stack< Integer > variable
20        Stack< Integer > integerStack = new Stack( 10 );
21
22        testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
23        testPop( "rawTypeStack1", rawTypeStack1 );
24        testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
25        testPop( "rawTypeStack2", rawTypeStack2 );
26        testPush( "integerStack", integerStack, integerElements );
27        testPop( "integerStack", integerStack );
28    } // end method testStacks
29

```

Instantiate generic class
Stack with raw type

Assign a Stack< Double >
to variable rawTypeStack2

Assign a Stack of raw type
to Stack< Integer >.
Legal but unsafe




```
30 // generic method pushes elements onto stack
31 public < T > void testPush( String name, Stack< T > stack,
32     T[] elements )
33 {
34     // push elements onto stack
35     try
36     {
37         System.out.printf( "\nPushing elements onto %s\n", name );
38
39         // push elements onto stack
40         for ( T element : elements )
41         {
42             System.out.printf( "%s ", element );
43             stack.push( element ); // push element onto stack
44         } // end for
45     } // end try
46     catch ( FullStackException fullStackException )
47     {
48         System.out.println();
49         fullStackException.printStackTrace();
50     } // end catch FullStackException
51 } // end method testPush
52
```



```
53 // generic method testPop pops elements from stack
54 public < T > void testPop( String name, Stack< T > stack )
55 {
56     // pop elements from stack
57     try
58     {
59         System.out.printf( "\nPopping elements from %s\n", name );
60         T popValue; // store element removed from stack
61
62         // remove elements from Stack
63         while ( true )
64         {
65             popValue = stack.pop(); // pop from stack
66             System.out.printf( "%s ", popValue );
67         } // end while
68     } // end try
69     catch( EmptyStackException emptyStackException )
70     {
71         System.out.println();
72         emptyStackException.printStackTrace();
73     } // end catch EmptyStackException
74 } // end method testPop
75
76 public static void main( String args[] )
77 {
78     RawTypeTest application = new RawTypeTest();
79     application.testStacks();
80 } // end main
81 } // end class RawTypeTest
```



Pushing elements onto rawTypeStack1

1.1 2.2 3.3 4.4 5.5 6.6

FullStackException: Stack is full, cannot push 6.6

```
at Stack.push(Stack.java:30)
at RawTypeTest.testPush(RawTypeTest.java:43)
at RawTypeTest.testStacks(RawTypeTest.java:22)
at RawTypeTest.main(RawTypeTest.java:79)
```

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:40)
at RawTypeTest.testPop(RawTypeTest.java:65)
at RawTypeTest.testStacks(RawTypeTest.java:23)
at RawTypeTest.main(RawTypeTest.java:79)
```

Pushing elements onto rawTypeStack2

1.1 2.2 3.3 4.4 5.5 6.6

FullStackException: Stack is full, cannot push 6.6

```
at Stack.push(Stack.java:30)
at RawTypeTest.testPush(RawTypeTest.java:43)
at RawTypeTest.testStacks(RawTypeTest.java:24)
at RawTypeTest.main(RawTypeTest.java:79)
```



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:40)
    at RawTypeTest.testPop(RawTypeTest.java:65)
    at RawTypeTest.testStacks(RawTypeTest.java:25)
    at RawTypeTest.main(RawTypeTest.java:79)
```

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

```
FullStackException: Stack is full, cannot push 11
    at Stack.push(Stack.java:30)
    at RawTypeTest.testPush(RawTypeTest.java:43)
    at RawTypeTest.testStacks(RawTypeTest.java:26)
    at RawTypeTest.main(RawTypeTest.java:79)
```

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:40)
    at RawTypeTest.testPop(RawTypeTest.java:65)
    at RawTypeTest.testStacks(RawTypeTest.java:27)
    at RawTypeTest.main(RawTypeTest.java:79)
```



```

RawTypeTest.java:20: warning: unchecked assignment
found    : Stack
required: Stack<java.lang.Integer>
    Stack< Integer > integerStack = new Stack( 10 );
                                   ^
RawTypeTest.java:22: 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:23: 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:24: 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 );
    ^
RawTypeTest.java:25: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
    testPop( "rawTypeStack2", rawTypeStack2 );
    ^
5 warnings

```

Fig. 18.13 | Warning message from the compiler.



18.8 **Wildcards** in Methods That Accept Type Parameters

- **Data structure `ArrayList`**
 - Dynamically resizable, array-like data structure
 - Method `add`
 - Method `toString`
- **Motivation for using wildcards**
 - Implement a generic method `sum`
 - Total the numbers in a collection
 - Receive a parameter of type `ArrayList< Number >`
 - Use method `doubleValue` of class `Number` to obtain the `Number`'s underlying primitive value as a `double` value



```

1 // Fig. 18.14: TotalNumbers.java
2 // Summing the elements of an ArrayList.
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 Number
10        // both Integers and Doubles, then display total
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
16
17        System.out.printf( "numberList contains: " );
18        System.out.printf( "Total of the elements in numberList: %n",
19            sum( numberList ) );
20    } // end main
21

```

Declare and initialize
array **numbers**

Declare and initialize **numberList**,
which stores **Number** objects

Add elements in **numbers** array
to ArrayList **numberList**

Invoke method **sum** to calculate the total
of the elements stored in **numberList**



```
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
```

Method `sum` accepts an `ArrayList` that stores `Number` objects

Use method `doubleValue` of class `Number` to obtain the `Number`'s underlying primitive value as a `double` value

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



18.8 Wildcards in Methods That Accept Type Parameters (Cont.)

- Implementing method `sum` with a wildcard type argument in its parameter
 - `Number` is the superclass of `Integer`
 - `ArrayList< Number >` is not a supertype of `ArrayList< Integer >`
 - Cannot pass `ArrayList< Integer >` to method `sum`
 - Use wildcard to create a more flexible version of `sum`
 - `ArrayList< ? extends Number >`
 - `?` Represents an “unknown type”
 - Unknown type argument must be either `Number` or a subclass of `Number`
 - Cannot use wildcard as a type name through method body



```
1 // Fig. 18.15: 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
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
```

Declare and create ArrayList
integerList to hold Integers

Invoke method sum to calculate the total
of the elements stored in integerList

Declare and create ArrayList
doubleList to hold Doubles



```

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
36 // both Integers and Doubles, then
37 Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
38 ArrayList< Number > numberList = new ArrayList< Number >();
39
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 // calculate total of stack elements
50 public static double sum( ArrayList< ? extends Number > list )
51 {
52     double total = 0; // initialize total
53

```

Invoke method `sum` to calculate the total of the elements stored in `doubleList`

Declare and create `ArrayList` `numberList` to hold `Numbers`

Invoke method `sum` to calculate the total of the elements stored in `numberList`

The `ArrayList` argument's element types are not directly known by the method, they are known to be at least of type `Number`



```
54     // calculate sum
55     for ( Number element : list )
56         total += element.doubleValue();
57
58     return total;
59 } // end method sum
60 } // end class WildcardTest
```

```
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
```



Common Programming Error 18.4

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.



18.9 Generics and Inheritance: Notes

- **Inheritance in generics**
 - **Generic class can be derived from non-generic class**
e.g., class `Object` is superclass of every generic class
 - **Generic class can be derived from another generic class**
e.g., `Stack` is a subclass of `Vector`
 - **Non-generic class can be derived from generic class**
e.g., `Properties` is a subclass of `Hashtable`
 - **Generic method in subclass can override generic method in superclass**
 - **If both methods have the same signature**

