Chapter 20
Generic Classes and Methods

Java How to Program, 10/e

OBJECTIVES

In this chapter you'll:

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

20.1 Introduction

- Detect type mismatches at *compile time*—known as compile-time type safety.
- Generic methods and generic classes provide the means to create type safe general models.

20.2 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. 20.2.

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

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

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

```

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

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

      System.out.println();
   }

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

      System.out.println();
   }

```

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

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

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

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

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

20.3 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 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.

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

public class GenericMethodTest
{
   public static void main(String[] args)
   {
      // create arrays of Integer, Double and Character
      Integer[] 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.printf("Array integerArray contains:%n");
      printArray(integerArray); // pass an Integer array
      System.out.printf("%nArray doubleArray contains:%n");
      printArray(doubleArray); // pass a Double array
      System.out.printf("%nArray characterArray contains:%n");
      printArray(characterArray); // pass a Character array
   }

```

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

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

       System.out.println();
    }
} // 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. 20.3 | Printing array elements using generic method `printArray`. (Part 2 of 2.)

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

- All generic method declarations have a type-parameter section (< T > in this example) delimited by angle brackets that precedes the method's return type.
- Each type-parameter section contains one or more 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.

Good Programming Practice 20.1

The letters `T` *(for "type"),* `E` *(for "element"),* `K` *(for "key") and* `V` *(for "value") are commonly used as type parameters. For other common ones, see* `http://docs.oracle.com/javase/tutorial/java/generics/types.html`.

Common Programming Error 20.1

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

Common Programming Error 20.2

If the compiler doesn't find a method declaration that matches a method call exactly, but does find two or more methods that can satisfy the method call, a compilation error occurs. For the complete details of resolving calls to overloaded and generic methods, see `http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12`.

20.3 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. 20.4—there is only *one* copy of this code, which is used for all `printArray` calls in the example.

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

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

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

20.4 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.
- Generic interfaces enable you to specify, with a single interface declaration, a set of related types.

20.4 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`).

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

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

```

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

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

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

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

       return max; // returns the largest object
    }
 } // 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. 20.5 | Generic method `maximum` with an upper bound on its type parameter. (Part 2 of 2.)

20.4 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 type parameter's upper bound.
- 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 `Comparable<T>` restriction is important, because not all objects can be compared.

- More than one bounding type can be given as follows:
- T extends Comparable & Serializable
- If one of the bounds is a class, it should be given first.

20.4 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.

```java
public static Comparable maximum(Comparable x, Comparable y, Comparable z)
{
   Comparable max = x; // assume x is initially the largest

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

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

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

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

20.5 Overloading Generic Methods

- *A generic method may be overloaded like any other method.*
- 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 best matches the method name and the argument types specified in the call—an error occurs if two or more overloaded methods both could be considered best matches.

20.6 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 ty*pe-independent* manner.
- These classes are known as parameterized classes or parameterized types because they accept one or more type parameters.

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

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

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

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

```

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

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

    // 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 class Stack<T>
```

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

```java
// Fig. 20.8: EmptyStackException.java
// EmptyStackException class declaration.
public class EmptyStackException extends RuntimeException
{
   // no-argument constructor
   public EmptyStackException()
   {
      this("Stack is empty");
   }

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

Fig. 20.8 | `EmptyStackException` class declaration.

```java
// Fig. 20.9: 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<>(5);
      Stack<Integer> integerStack = new Stack<>();

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

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

```

Fig. 20.9 | Stack generic class test program. (Part 1 of 6.)

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

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

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

```

Fig. 20.9 | Stack generic class test program. (Part 2 of 6.)

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

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

```

Fig. 20.9 | Stack generic class test program. (Part 3 of 6.)

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

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

```

Fig. 20.9 | Stack generic class test program. (Part 4 of 6.)

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

Fig. 20.9 | 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. 20.9 | `Stack` generic class test program. (Part 6 of 6.)

20.6 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.

```java
// Fig. 20.10: 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<>(5);
      Stack<Integer> integerStack = new Stack<>();

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

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

```

Fig. 20.10 | Passing generic **Stack** objects to generic methods. (Part 1 of 4.)

```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
      }
   }

   // 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

```

Fig. 20.10 | Passing generic `Stack` objects to generic methods. (Part 2 of 4.)

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

Fig. 20.10 | 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. 20.10 | Passing generic `Stack` objects to generic methods. (Part 4 of 4.)

20.8 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` has 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 Java versions.
  - Raw-type operations are unsafe and could lead to exceptions.
    ```

```java
// Fig. 20.11: RawTypeTest.java
// Raw type test program.
public class RawTypeTest
{
   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};

      // Stack of raw types assigned to Stack of raw types variable
      Stack rawTypeStack1 = new Stack(5);

      // Stack<Double> assigned to Stack of raw types variable
      Stack rawTypeStack2 = new Stack<Double>(5);

      // Stack of raw types assigned to Stack<Integer> variable
      Stack<Integer> integerStack = new Stack(10);

      testPush("rawTypeStack1", rawTypeStack1, doubleElements);
      testPop("rawTypeStack1", rawTypeStack1);
      testPush("rawTypeStack2", rawTypeStack2, doubleElements);
      testPop("rawTypeStack2", rawTypeStack2);
      testPush("integerStack", integerStack, integerElements);
      testPop("integerStack", integerStack);
   }
```

Fig. 20.11 | Raw-type test program. (Part 1 of 4.)

```java

   // generic method pushes elements onto 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
      }
   }

   // generic method testPop pops elements from 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

```

Fig. 20.11 | Raw-type test program. (Part 2 of 4.)

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

Fig. 20.11 | 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. 20.11 | Raw-type test program. (Part 4 of 4.)

20.7 Raw Types (cont.)

- Figure 20.12 shows the warning messages generated by the compiler when the file `RawTypeTest.java` (Fig. 20.11) 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);
                                 ^
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);
             ^
```

Fig. 20.12 | Warning messages from the compiler. (Part 1 of 2.)

```
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);
              ^
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. 20.12 | Warning messages from the compiler. (Part 2 of 2.)

Things to note

- **Generics and Subtyping**
- In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

Things to note

- **A Generic Class is Shared by All Its Invocations**
- What does the following code fragment print?

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

- You might be tempted to say false, but you'd be wrong. It prints true, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.

Things to note

- **Casts and InstanceOf**
- Another implication of the fact that a generic class is shared among all its instances, is that it usually makes no sense to ask an instance if it is an instance of a particular invocation of a generic type:

```
Collection cs = new ArrayList<String>();
// Illegal.
if (cs instanceof Collection<String>) { ... }
```

similarly, a cast such as

```
// Unchecked warning,
Collection<String> cstr = (Collection<String>) cs;
```

gives an unchecked warning, since this isn't something the runtime system is going to check for you.

Things to note

- **Arrays**
- The component type of an array object may not be a type variable or a parameterized type, unless it is an (unbounded) wildcard type.You can declare array *types* whose element type is a type variable or a parameterized type, but not array *objects*.

```
// Not really allowed.
List<String>[] lsa = new List<String>[10];
```

20.8 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.

```java
// Fig. 20.13: TotalNumbers.java
// Totaling the numbers in an ArrayList<Number>.
import java.util.ArrayList;

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

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

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

```

Fig. 20.13 | Totaling the numbers in an ArrayList<Number>.

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

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

       return total;
    }
 } // end class TotalNumbers
```

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

Fig. 20.13 | Totaling the numbers in an `ArrayList<Number>`.

20.8 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`.

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

- Given that method `sum` can total the elements of an `ArrayList` of `Number`s, you might expect that the method would also work for `ArrayList`s that contain elements of only one numeric type, such as `ArrayList<Integer>`.
- Modified class `TotalNumbers` to create an `ArrayList-` of `Integer`s 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 TotalNumbersErrors cannot be applied to (java.util.ArrayList<java.lang.Integer>)`
- Although `Number` is the superclass of `Integer`, the compiler doesn't 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>`.

20.8 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. 20.14, 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`.

```java
// Fig. 20.14: WildcardTest.java
// Wildcard test program.
import java.util.ArrayList;

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

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

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

```

Fig. 20.14 | Wildcard test program. (Part 1 of 4.)

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

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

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

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

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

```

Fig. 20.14 | Wildcard test program. (Part 2 of 4.)

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

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

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

      return total;
   }
} // end class WildcardTest
```

Fig. 20.14 | 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. 20.14 | Wildcard test program. (Part 4 of 4.)

20.8 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.
    ```