20.6 Generic Classes

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. We can then instantiate *type-specific* objects of the generic class. Generics provide a nice opportunity for software reusability.

Once you have a generic class, you can use a simple, concise notation to indicate the type(s) that should be used in place of the class's type parameter(s). At compilation time, the compiler ensures the *type safety* of your code and uses the *erasure* techniques described in Sections 20.3–20.4 to enable your client code to interact with the generic class.

One generic `Stack` class, for example, could be the basis for creating many logical `Stack` classes (e.g., "`Stack` of `Double`," "`Stack` of `Integer`," "`Stack` of `Character`," "`Stack` of `Employee`"). These classes are known as **parameterized classes** or **parameterized types** because they accept one or more type parameters. Recall that type parameters represent only *reference types*, which means the `Stack` generic class cannot be instantiated with primitive types. However, we can instantiate a `Stack` that stores objects of Java's type-wrapper classes and allow Java to use *autoboxing* to convert the primitive values into objects. Recall that autoboxing occurs when a value of a primitive type (e.g.,

`int`) is pushed onto a `Stack` that contains wrapper-class objects (e.g., `Integer`). *Auto-unboxing* occurs when an object of the wrapper class is popped off the `Stack` and assigned to a primitive-type variable.

Implementing a Generic `Stack` Class

Figure 20.7 declares a generic `Stack` class for demonstration purposes—the `java.util` package already contains a generic `Stack` class. A generic class declaration looks like a non-generic one, but the class name is followed by *a type-parameter section* (line 6). In this case, type parameter `E` represents the *element* type the `Stack` will manipulate. As with generic methods, the type-parameter section of a generic class can have one or more type parameters separated by commas. (You'll create a generic class with two type parameters in Exercise 20.8.) Type parameter `E` is used throughout the `Stack` class declaration to represent the element type. This example implements a `Stack` as an `ArrayList`.

```
1   // Fig. 20.7: Stack.java
2   // Stack generic class declaration.
3   import java.util.ArrayList;
4   import java.util.NoSuchElementException;
5
6   public class Stack<E> {
7      private final ArrayList<E> elements; // Array
```

```
                        8
 9      // no-argument constructor creates a stack of
             10      public Stack() {
   11          this(10); // default stack size
                    12      }
                        13
   14      // constructor creates a stack of the specifi
        15      public Stack(int capacity) {
   16         int initCapacity = capacity > 0 ? capacity
   17         elements = new ArrayList<E>(initCapacity);
                    18      }
                        19
         20      // push element onto stack
      21      public void push(E pushValue){
   22         elements.add(pushValue); // place pushValu
                    23      }
                        24
   25      // return the top element if not empty; else
             26      public E pop() {
   27         if (elements.isEmpty()) { // if stack is e
   28            throw new NoSuchElementException("Stack
                   29         }
                        30
   31         // remove and return top element of Stack
   32         return elements.remove(elements.size() - 1
                    33      }
                      34   }
```

Fig. 20.7

Stack generic class declaration.

Class `Stack` declares variable `elements` as an `ArrayList<E>` (line 7). This `ArrayList` will store the

`Stack`'s elements. As you know, an `ArrayList` can grow dynamically, so objects of our `Stack` class can also grow dynamically. The `Stack` class's no-argument constructor (lines 10–12) invokes the one-argument constructor (lines 15–18) to create a `Stack` in which the underlying `ArrayList` has a capacity of `10` elements. The one-argument constructor can also be called directly to create a `Stack` with a specified initial capacity. Line 16 validates the constructor's argument. Line 17 creates the `ArrayList` of the specified capacity (or 10 if the capacity was invalid).

Method `push` (lines 21–23) uses `ArrayList` method `add` to append the pushed item to the end of the `ArrayList elements`. The last element in the `ArrayList` represents the stack's *top*.

Method `pop` (lines 26–33) first determines whether an attempt is being made to pop an element from an empty `Stack`. If so, line 28 throws a `NoSuchElementException` (package `java.util`). Otherwise, line 32 returns the `Stack`'s top element by removing the underlying `ArrayList`'s last element.

As with generic methods, when a generic class is compiled, the compiler performs *erasure* on the class's type parameters and replaces them with their upper bounds. For class `Stack` (Fig. 20.7), no upper bound is specified, so the default upper bound, `Object`, is used. The scope of a generic class's type parameter is the entire class. However, type parameters *cannot* be used in a class's `static` variable declarations.

Testing the Generic `Stack` *Class*

Now, let's consider the application (Fig. 20.8) that uses the `Stack` generic class (Fig. 20.7). Lines 11–12 in Fig. 20.8 create and initialize variables of type `Stack<Double>` (pronounced "`Stack` of `Double`") and `Stack<Integer>` (pronounced "`Stack` of `Integer`"). The types `Double` and `Integer` are known as the `Stack`'s **type arguments**. The compiler uses them to replace the type parameters so that it can perform type checking and insert cast operations as necessary. We'll discuss the cast operations in more detail shortly. Lines 11– 12 instantiate `doubleStack` with a capacity of `5` and `integerStack` with a capacity of 10 (the default). Lines 15–16 and 19–20 call methods `testPushDouble` (lines 24–33), `test-PopDouble` (lines 36–52), `testPushInteger` (lines 55–64) and `testPopInteger` (lines 67–83), respectively, to demonstrate the two `Stack`s in this example.

```
1   // Fig. 20.8: StackTest.java
2   // Stack generic class test program.
3   import java.util.NoSuchElementException;
4
5   public class StackTest {
6      public static void main(String[] args) {
7         double[] doubleElements = {1.1, 2.2, 3.3,
8         int[] integerElements = {1, 2, 3, 4, 5, 6,
9
10        // Create a Stack<Double> and a Stack<Inte
11        Stack<Double> doubleStack = new Stack<>(5)
12        Stack<Integer> integerStack = new Stack<>(
13
14        // push elements of doubleElements onto do
```

```
15          testPushDouble(doubleStack, doubleElements
16          testPopDouble(doubleStack); // pop from do
                      17
18          // push elements of integerElements onto i
19          testPushInteger(integerStack, integerEleme
20          testPopInteger(integerStack); // pop from
                    21      }
                      22
  23      // test push method with double stack
   24      private static void testPushDouble(
25          Stack<Double> stack, double[] values) {
26          System.out.printf("%nPushing elements onto
                      27
       28         // push elements to Stack
    29          for (double value : values) {
30             System.out.printf("%.1f ", value);
31             stack.push(value); // push onto doubleS
                  32          }
                    33      }
                      34
  35      // test pop method with double stack
36      private static void testPopDouble(Stack<Doubl
     37          // pop elements from stack
                 38          try {
39             System.out.printf("%nPopping elements f
40             double popValue; // store element remov
                      41
 42             // remove all elements from Stack
          43             while (true) {
44                popValue = stack.pop(); // pop from
45                System.out.printf("%.1f ", popValue)
                 46             }
                   47          }
48          catch(NoSuchElementException noSuchElement
       49             System.err.println();
50             noSuchElementException.printStackTrace(
                  51          }
                    52      }
                      53
 54      // test push method with integer stack
```

```
   55      private static void testPushInteger(
56          Stack<Integer> stack, int[] values) {
57          System.out.printf("%nPushing elements onto
                        58
       59          // push elements to Stack
      60          for (int value : values) {
 61             System.out.printf("%d ", value);
62             stack.push(value); // push onto integer
                   63          }
                     64      }
                        65
  66      // test pop method with integer stack
67      private static void testPopInteger(Stack<Inte
      68          // pop elements from stack
                  69          try {
70             System.out.printf("%nPopping elements f
71             int popValue; // store element removed
                           72
 73             // remove all elements from Stack
          74             while (true) {
75                popValue = stack.pop(); // pop from
76                System.out.printf("%d ", popValue);
                  77             }
                    78          }
79          catch(NoSuchElementException noSuchElement
       80             System.err.println();
81             noSuchElementException.printStackTrace(
                   82          }
                     83      }
                       84   }
```

```
        Pushing elements onto doubleStack
               1.1 2.2 3.3 4.4 5.5
        Popping elements from doubleStack
               5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, can
               at Stack.pop(Stack.java:28)
        at StackTest.testPopDouble(StackTest.java:44)
```

```
            at StackTest.main(StackTest.java:16)

         Pushing elements onto integerStack
                1 2 3 4 5 6 7 8 9 10
        Popping elements from integerStack
               10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, can
               at Stack.pop(Stack.java:28)
        at StackTest.testPopInteger(StackTest.java:75
            at StackTest.main(StackTest.java:20)
```

Fig. 20.8

Stack generic class test program.

Methods `testPushDouble` and `testPopDouble`

Method `testPushDouble` (lines 24–33) invokes method `push` (line 31) to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 from array `doubleElements` onto `doubleStack`. *Autoboxing* occurs in line 31 when the program tries to push a primitive `double` value onto the `doubleStack`, which stores only references to `Double` objects.

Method `testPopDouble` (lines 36–52) invokes `Stack` method `pop` (line 44) in an infinite loop (lines 43–46) to remove all the values from the stack. The output shows that

the values indeed `pop` off in last-in, first-out order (the defining characteristic of stacks). When the loop attempts to `pop` a sixth value, the `doubleStack` is empty, so `pop` throws a `NoSuchElementException`, which causes the program to proceed to the `catch` block (lines 48–51). The stack trace indicates the exception that occurred and shows that method `pop` generated the exception at line 28 of the file `Stack.java` (Fig. 20.7). The trace also shows that `pop` was called by `StackTest` method `testPopDouble` at line 44 (Fig. 20.8) of `Stack-Test.java` and that method `testPopDouble` was called from method `main` at line 16 of `StackTest.java`. This information enables you to determine the methods that were on the method-call stack at the time the exception occurred. Because the program catches the exception, the exception is considered to have been handled and the program can continue executing.

Auto-unboxing occurs in line 44 when the program assigns the `Double` object popped from the stack to a `double` primitive variable. Recall from Section 20.4 that the compiler inserts casts to ensure that the proper types are returned from generic methods. After erasure, `Stack` method `pop` returns type `Object`, but the client code in `testPopDouble` expects to receive a `double` when method `pop` returns. So the compiler inserts a `Double` cast, as in

```
popValue = (Double) stack.pop();
```

The value assigned to `popValue` will be *unboxed* from the

`Double` object returned by `pop`.

Methods `testPushInteger` and `testPopInteger`

Method `testPushInteger` (lines 55–64) invokes `Stack` method `push` to place values onto `integerStack` until it's full. Method `testPopInteger` (lines 67–83) invokes `Stack` method `pop` to remove values from `integerStack`. Once again, the values are popped in last-in, first-out order. During *erasure*, the compiler recognizes that the client code in method `testPopInteger` expects to receive an `int` when method `pop` returns. So the compiler inserts an `Integer` cast, as in

```
popValue = (Integer) stack.pop();
```

The value assigned to `popValue` will be unboxed from the `Integer` object returned by `pop`.

Creating Generic Methods to Test Class `Stack<E>`

The code in methods `testPushDouble` and `testPushInteger` 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. Figure 20.9 declares generic method `testPush` (lines 24–33) to perform the same tasks as `testPushDouble` and `testPushInteger` in Fig. 20.8—that is, `push` values onto a `Stack<E>`. Similarly, generic method `testPop` (Fig. 20.9, lines 36–52) performs the same tasks as `testPopDouble` and `testPopInteger` in Fig. 20.8—that is, `pop` values off a `Stack<E>`. The output of Fig. 20.9 precisely matches that of Fig. 20.8.

```
1   // Fig. 20.9: StackTest2.java
2   // Passing generic Stack objects to generic meth
3   import java.util.NoSuchElementException;
4
5   public class StackTest2 {
6      public static void main(String[] args) {
7         Double[] doubleElements = {1.1, 2.2, 3.3,
8         Integer[] integerElements = {1, 2, 3, 4, 5
9
10        // Create a Stack<Double> and a Stack<Inte
11        Stack<Double> doubleStack = new Stack<>(5)
12        Stack<Integer> integerStack = new Stack<>(
13
14        // push elements of doubleElements onto do
15        testPush("doubleStack", doubleStack, doubl
16        testPop("doubleStack", doubleStack); // po
17
18        // push elements of integerElements onto i
```

```
19         testPush("integerStack", integerStack, int
20         testPop("integerStack", integerStack); //
                  21      }
                     22
23      // generic method testPush pushes elements on
24      public static <E> void testPush(String name ,
            25         E[] elements) {
26         System.out.printf("%nPushing elements onto
                     27
     28         // push elements onto Stack
     29         for (E element : elements) {
30            System.out.printf("%s ", element);
31            stack.push(element); // push element on
                  32         }
                    33      }
                        34
35      // generic method testPop pops elements from
36      public static <E> void testPop(String name, S
      37         // pop elements from stack
                 38         try {
39            System.out.printf("%nPopping elements f
40            E popValue; // store element removed fr
                        41
 42            // remove all elements from Stack
          43            while (true) {
    44               popValue = stack.pop();
45               System.out.printf("%s ", popValue);
                  46            }
                    47         }
48         catch(NoSuchElementException noSuchElement
       49            System.out.println();
50            noSuchElementException.printStackTrace(
                  51         }
                    52      }
                      53   }
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
```

```
          Popping elements from doubleStack
                5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, can
                at Stack.pop(Stack.java:28)
          at StackTest2.testPop(StackTest2.java:44)
            at StackTest2.main(StackTest2.java:16)

         Pushing elements onto integerStack
                1 2 3 4 5 6 7 8 9 10
         Popping elements from integerStack
               10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, can
                at Stack.pop(Stack.java:28)
          at StackTest2.testPop(StackTest2.java:44)
            at StackTest2.main(StackTest2.java:20)
```

Fig. 20.9

Passing generic Stack objects to generic methods.

Lines 11–12 create the `Stack<Double>` and `Stack<Integer>` objects, respectively. Lines 15–16 and 19–20 invoke generic methods `testPush` and `testPop` to test the `Stack` objects. Type parameters can represent only reference types, so to be able to pass arrays `doubleElements` and `integerElements` to generic method `testPush`, the arrays declared in lines 7–8 must be declared with the wrapper types `Double` and `Integer`. When these arrays are initialized with primitive values, the compiler *autoboxes* each primitive value.

Generic method `testPush` (lines 24–33) uses type parameter `E` (specified at line 24) to represent the data type stored in the `Stack<E>`. The generic method takes three arguments—a `String` that represents the name of the `Stack<E>` object for output purposes, a reference to an object of type `Stack<E>` and an array of type `E`—the type of elements that will be `push`ed onto `Stack<E>`. The compiler enforces *consistency* between the type of the `Stack` and the elements that will be pushed onto the `Stack` when `push` is invoked, which is the real value of the generic method call. Generic method `testPop` (lines 36–52) takes two arguments—a `String` that represents the name of the `Stack<E>` object for output purposes and a reference to an object of type `Stack<E>`.