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

Let’s consider a generic method in which type parameters are used in the return type and in the parameter list (Fig. 20.5). The application uses a generic method `maximum` to determine and return the largest of its three arguments of the same type. Unfortunately, *the relational operator* `>` *cannot be used with reference types*. However, it’s possible to compare two objects of the same class if that class implements the generic **interface** `Comparable<T>` (from 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. `Comparable<T>` objects have a `compareTo` **method**. For example, two `Integer` objects, `integer1` and `integer2`, can be compared with the expression:

```
integer1.compareTo(integer2)
```

When you declare a class that implements `Comparable<T>`,

you must define method `compareTo` such that it compares the contents of two objects of that class and returns the comparison results. Method `compareTo` *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`.

For example, class `Integer`'s `compareTo` method compares the `int` values stored in two `Integer` objects. 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`). We discussed those methods in Chapter 16. In this example, we'll use method `compareTo` in method `maximum` to help determine the largest value.

```
   1   // Fig. 20.5: MaximumTest.java
2   // Generic method maximum returns the largest of
                    3
     4   public class MaximumTest {
5      public static void main(String[] args) {
6         System.out.printf("Maximum of %d, %d and %
       7            maximum(3, 4, 5));
8         System.out.printf("Maximum of %.1f, %.1f a
9            6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
10        System.out.printf("Maximum of %s, %s and %
11           "apple", "orange", maximum("pear", "app
                  12      }
                     13
14     // determines the largest of three Comparable
15     public static <T extends Comparable<T>> T max
16        T max = x; // assume x is initially the la
                    17
```

```
      18          if (y.compareTo(max) > 0) {
19              max = y; // y is the largest so far
                    20          }
                         21
      22          if (z.compareTo(max) > 0) {
   23              max = z; // z is the largest
                    24          }
                         25
26          return max; // returns the largest object
                        27      }
                          28    }
```

```
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.

Generic Method `maximum` and Specifying a Type Parameter's Upper Bound

Generic method `maximum` (lines 15–27) uses type parameter `T` as its return type (line 15), as the type of method parameters

`x`, `y` and `z` (line 15), and as the type of local variable `max` (line 16). 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<T>` is known as the type parameter's **upper bound**. By default, `Object` is the upper bound, meaning that an object of any type can be used. 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 upper bound may be a comma-separated list that contains zero or one class and zero or more interfaces.

Method `maximum`'s type parameter is more restrictive than the one specified for `print-Array` in Fig. 20.3, which was able to output arrays containing any type of object. The `Comparable<T>` restriction is important, because not all objects can be compared. However, `Comparable<T>` objects are guaranteed to have a `compareTo` method.

Method `maximum` uses the same algorithm that we used in Section 6.4 to determine the largest of its three arguments. The method assumes that its first argument (`x`) is the largest and assigns it to local variable `max` (line 16 of Fig. 20.5). Next, the `if` statement at lines 18–20 determines whether `y` is greater than `max`. The condition invokes `y`'s `compareTo` method with the expression `y.compareTo(max)`, which returns a negative integer, `0` or a positive integer, to determine `y`'s relationship to `max`. If the return value of the `compareTo` is greater than `0`, then `y` is greater and is assigned to variable

`max`. Similarly, the `if` statement at lines 22–24 determines whether `z` is greater than `max` and, if so, assigns `z` to `max`. Then line 26 returns `max` to the caller.

Calling Method `maximum`

In `main`, line 7 calls `maximum` with the integers `3`, `4` and `5`. When the compiler encounters this call, it first looks for a `maximum` method that takes three arguments of type `int`. There's no such method, so the compiler looks for a generic method that can be used and finds generic method `maximum`. However, recall that the arguments to a generic method must be of a *reference type*. So the compiler autoboxes the three `int` values as `Integer` objects and specifies that the three `Integer` objects will be passed to `maximum`. Class `Integer` (package `java.lang`) implements the `Comparable<Integer>` interface such that method `compareTo` compares the `int` values in two `Integer` objects. Therefore, `Integer`s are valid arguments to method `maximum`. When the `Integer` representing the maximum is returned, we attempt to output it with the `%d` format specifier, which outputs an `int` primitive-type value. So `maximum`'s return value is output as an `int` value.

A similar process occurs for the three `double` arguments passed to `maximum` in line 9. Each `double` is autoboxed as a `Double` object and passed to `maximum`. Again, this is allowed because class `Double` (package `java.lang`) implements the `Comparable<Double>` interface. The

`Double` returned by `maximum` is output with the format specifier `%.1f`, which outputs a `double` primitive-type value. So `maximum`'s return value is auto-unboxed and output as a `double`. The call to `maximum` in line 11 receives three `String`s, which are also `Comparable<String>` objects. We intentionally placed the largest value in a different position in each method call (lines 7, 9 and 11) to show that the generic method always finds the maximum value, regardless of its position in the argument list.

Erasure and the Upper Bound of a Type Parameter

When the compiler translates method `maximum` into bytecodes, it uses erasure to replace the type parameters with actual types. In Fig. 20.3, all generic types were replaced with type `Object`. Actually, all type parameters are replaced with the *upper bound* of the type parameter, which is specified in the type-parameter section. Figure 20.6 simulates the erasure of method maximum's types by showing the method's source code after the type-parameter section is removed and type parameter `T` is replaced with the upper bound, `Comparable`, throughout the method declaration. The erasure of `Comparable<T>` is simply `Comparable`.

```
1   public static Comparable maximum(Comparable x, C
              2      Comparable z) {
                          3
4      Comparable max = x; // assume x is initially
```

```
                        5
      6      if (y.compareTo(max) > 0) {
   7         max = y; // y is the largest so far
                        8      }
                        9
     10      if (z.compareTo(max) > 0) {
  11          max = z; // z is the largest
                       12      }
                       13
 14      return max; // returns the largest object
                       15   }
```

Fig. 20.6

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

After erasure, method `maximum` specifies that it returns type `Comparable`. However, the calling method does not expect to receive a `Comparable`. It expects to receive an object of the same type that was passed to `maximum` as an argument —`Integer`, `Double` or `String` in this example. 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. Thus, the call to `maximum` in line 7 (Fig. 20.5) is preceded by an `Integer` cast, as in

```
(Integer) maximum(3, 4, 5)
```

the call to `maximum` in line 9 is preceded by a `Double` cast, as in

```
   (Double) maximum(6.6, 8.8, 7.7)
```

and the call to `maximum` in line 11 is preceded by a `String` cast, as in

```
   (String) maximum("pear", "apple", "orange")
```

In each case, the type of the cast for the return value is *inferred* from the types of the method arguments in the particular method call, because, according to the method declaration, the return type and the argument types match. Without generics, you'd be responsible for implementing the cast operation.