4.7 IMPLEMENTING GENERIC COMPONENTS USING JAVA 1.5 GENERICS

We have already seen that Java 1.5 supports generic classes, and these classes are easy to use. However, writing generic classes requires a little more work. In this section, we illustrate the basics of how generic classes and methods are written. We do not attempt to cover all the constructs of the language, which are quite complex and sometimes tricky. Instead, we show the syntax and idioms that are used throughout this book.

4.7.1 Simple Generic Classes and Interfaces

Figure 4.28 shows a generic version of the `MemoryCell` class previously depicted in Figure 4.21. Here, we have changed the name to `GenericMemoryCell`, since neither class is in a package, and thus the names cannot be the same.

```
1 public class GenericMemoryCell<AnyType>
2 {
3     public AnyType read( )
4       { return storedValue; }
5     public void write( AnyType x )
6       { storedValue = x; }
7
8     private AnyType storedValue;
9 }
```

Figure 4.28 Generic implementation of the `MemoryCell` class.

```
1 package java.lang;
2
3 public interface Comparable<AnyType>
4 {
5     public int compareTo( AnyType other );
6 }
```

Figure 4.29 `Comparable` interface, Java 1.5 version which is generic.

When a generic class is specified, the class declaration includes one or more *type parameters*, enclosed in angle brackets <> after the class name.

When a generic class is specified, the class declaration includes one or more *type parameters*, enclosed in angle brackets <> after the class name. Line 1 shows that the `GenericMemoryCell` takes one type parameter. In this instance, there are no restrictions on the type parameter, so the user can create types such as `GenericMemoryCell<String>`, `GenericMemoryCell<Integer>`, but not `GenericMemoryCell<int>`. Inside the `GenericMemoryCell` class declaration, we can declare fields of the generic type, and methods that use the generic type as a parameter or return type.

Interfaces can also be declared as generic.

Interfaces can also be declared as generic. For example, prior to Java 1.5 the `Comparable` interface was not generic, and its `compareTo` method took an `Object` as the parameter. As a result any reference variable passed to the `compareTo` method would compile, even if the variable was not a sensible type, and only at runtime would the error be reported as a `ClassCastException`. In Java 1.5, the `Comparable` class is generic, as shown in Figure 4.29. The `String` class, for instance, now implements `Comparable<String>`, and has a `compareTo` method that takes a `String` as a parameter. By making the class generic, many of the errors that were previously only reported at runtime become compile-time errors.

4.7.2 Wildcards With Bounds

In Figure 4.13 we saw a static method that computes the total area in an array of `Shape`s. Suppose we want to rewrite the method, so that it works with a parameter that is `List<Shape>`. Because of the enhanced for loop, the code should be identical, and the resulting code is shown in Figure 4.30. If we pass a `List<Shape>`, or `ArrayList<Shape>`, or `LinkedList<Shape>`, the code works. However, what happens if we pass a `List<Square>`? The answer depends on whether a `List<Square>` *IS-A* `List<Shape>`. Recall from Section 4.1.10, that the technical term for this is whether we have covariance.

Generic collections are not covariant.

In Java, as we mentioned in Section 4.1.10, arrays are covariant. So `Square[]` *IS-A* `Shape[]`. One the one hand, consistency would suggest that if arrays are covariant, then collections should be covariant too. On the other hand, as we saw in Section 4.1.10, the covariance of arrays leads to code that compiles but then generates a runtime exception (an `ArrayStoreException`). Because the entire reason to have generics is to generate compiler errors rather than runtime exceptions for type mismatches, generic collections are not covariant. As a result, we cannot pass a `List<Square>` as a parameter to the method in Figure 4.30.

What we are left with is that generics (and the generic collections) are not covariant (which makes sense), but arrays are. Without additional syntax, users would tend to avoid collections, because the lack of covariance makes the code less flexible.

```
 1 public static double totalArea( List<Shape> arr )
 2 {
 3     double total = 0;
 4
 5     for( Shape s : arr )
 6         if( s != null )
 7             total += s.area( );
 8
 9     return total;
10 }
```

Figure 4.30 `totalArea` method that does not work if passed a `List<Square>`

```
 1 public static double totalArea( List<? extends Shape> arr )
 2 {
 3     double total = 0;
 4
 5     for( Shape s : arr )
 6         if( s != null )
 7             total += s.area( );
 8
 9     return total;
10 }
```

Figure 4.31 `totalArea` method revised with wildcards that works if passed a `List<Square>`

Wildcards are used to express subclasses (or superclasses) of parameter types.

Java 1.5 makes up for this with *wildcards*. Wildcards are used to express subclasses (or superclasses) of parameter types. Figure 4.31 illustrates the use of wildcards with a bound to write a `totalArea` method that takes as parameter a `List<T>`, where `T` *IS-A* `Shape`. Thus, `List<Shape>`, `List<Square>`, and `ArrayList<Square>` are all acceptable parameters. Wildcards can also be used without a bound (in which case `extends Object` is presumed), or with `super` instead of `extends` (to express superclass rather than subclass) and there are also some other syntax uses that we do not discuss here.

4.7.3 Generic Static Methods

In some sense, the `totalArea` method in Figure 4.31 is generic, since it works for different types. But there is no specific type parameter list, as was done in the `GenericMemoryCell` class declaration. Sometimes the specific type is important, perhaps because one of the following reasons apply:

1. the type is used as the return type
2. the type is used in more than one parameter type
3. the type is used to declare a local variable

If so, then an explicit generic method, with type parameters must be declared.

For instance, Figure 4.32 illustrates a generic static method that performs a sequential search for value `x` in array `arr`. By using a generic method instead of a non-generic method that uses `Object` as the parameter types, we can get compile-time errors if searching for an `Apple` in array of `Shape`s.

The generic method looks much like the generic class, in that the type parameter list uses the same syntax. The type parameters in a generic method precedes the return type.

The generic method looks much like the generic class, in that the type parameter list uses the same syntax. The type list in a generic method precedes the return type.

```
1 public static <AnyType>
2 boolean contains( AnyType [ ] arr, AnyType x )
3 {
4     for( AnyType val : arr )
5         if( x.equals( val ) )
6             return true;
7
8     return false;
9 }
```

Figure 4.32 Generic static method to search an array

4.7.4 Type Bounds

The type bound is specified inside the angle brackets <>.

Suppose we want to write a `findMax` routine. Consider the code in Figure 4.33. This code cannot work, because the compiler cannot prove that the call to `compareTo` at line 6 will work. `compareTo` is guaranteed to exist only if `AnyType` is `Comparable`. We can solve this problem by using a *type bound*. The type bound is specified inside the angle brackets <>, and specifies properties that the parameter types must have. A naive attempt is to rewrite the signature as

```
public static <AnyType extends Comparable> ...
```

This is naive because as we know, the `Comparable` interface is now generic. Although this code would compile, a better attempt would be

```
public static <AnyType extends Comparable<AnyType>> ...
```

However, this attempt is not satisfactory. To see the problem, suppose `Shape` implements `Comparable<Shape>`. Suppose `Square` extends `Shape`. Then all we know is that `Square` implements `Comparable<Shape>`. Thus, a `Square` *IS-A* `Comparable<Shape>`, but it *IS-NOT-A* `Comparable<Square>`!

As a result, what we need to say is that `AnyType` *IS-A* `Comparable<T>` where `T` is a superclass of `AnyType`. Since we do not need to know the exact type `T`, we can use a wildcard. The resulting signature is

```
public static <AnyType extends Comparable<? super AnyType>>
```

```
1  public static <AnyType> AnyType findMax( AnyType [ ] a )
2  {
3      int maxIndex = 0;
4
5      for( int i = 1; i < a.length; i++ )
6          if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
7              maxIndex = i;
8
9      return a[ maxIndex ];
10 }
```

Figure 4.33 Generic static method to find largest element in an array that does not work.

```
1  public static <AnyType extends Comparable<? super AnyType>>
2  AnyType findMax( AnyType [ ] a )
3  {
4      int maxIndex = 0;
5
6      for( int i = 1; i < a.length; i++ )
7          if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
8              maxIndex = i;
9
10     return a[ maxIndex ];
11 }
```

Figure 4.34 Generic static method to find largest element in an array. This illustrates a bounds on the type parameter.

Figure 4.34 shows the implementation of `findMax`. The compiler will only accept arrays of types `T` such that `T` implements the `Comparable<S>` interface, where `T` *IS-A* `S`. Certainly the bounds declaration looks like a mess. Fortunately, we won't see anything more complicated than this idiom.

4.7.5 Type Erasure

Generic classes are converted by the compiler to non-generic classes by a process known as *type erasure*.

Generic types, for the most part, are constructs in the Java language, but not in the Virtual Machine. Generic classes are converted by the compiler to non-generic classes by a process known as *type erasure*. The simplified version of what happens is that the compiler generates a *raw class*, with the same name as the generic class with the type parameters removed. The type variables are replaced with their bounds, and when calls are made to generic methods that have an erased return type, casts are inserted automatically. If a generic class is used without a type parameter, the raw class is used.

Generics to not make the code faster. It does make the code more type-safe at compile time.

One important consequence of type erasure is that the generated code is not much different than the code that programmers have been writing prior to generics, and in fact is not any faster. The significant benefit is that the programmer does not have to place casts in the code, and the compiler will do significant type checking.

4.7.6 Restrictions on Generics

There are numerous restrictions on generic types. Every one of the restrictions listed below is required because of type erasure.

Primitive Types

Primitive types cannot be used for a type parameter.

Primitive types cannot be used for a type parameter. Thus `List<int>` is illegal. You must use wrapper classes.

`instanceOf` *Tests*

`instanceOf` tests and type casts only work with the raw type. Thus,

`instanceOf` tests and type casts only work with the raw type.

```
List<Integer> list1 = new ArrayList<Integer>( );
list1.add( 4 );
List<String> list2 = (List<String>) list1;
String s = list2.get( 0 );
```

compiles, but with a warning. At runtime, the typecast succeeds, since all types are `List`. Eventually, a runtime error will result at the last line because the call to `get` tries to return a `String`, but it cannot.

Static Contexts

In a generic class, static methods and fields cannot refer to the class' type variables, since after erasure, there are no type variables. Further, since there is really only one raw class, static fields are shared amongst the class' generic instantations.

Static methods and fields cannot refer to the class' type variables. Static fields are shared amongst the class' generic instantiations.

Instantiation of Generic Types

It is illegal to create an instance of a generic type. If `T` is a type variable, the statement

It is illegal to create an instance of a generic type.

```
T obj = new T( );          // Right-hand side is illegal
```

is illegal. `T` is replaced by its bounds, which could be `Object` (or even an abstract class), so the call to `new` cannot make sense.

Generic Array Objects

It is illegal to create a array of a generic type. If `T` is a type variable, the statement

It is illegal to create a array of a generic type.

```
T [ ] arr = new T[ 10 ];  // Right-hand side is illegal
```

is illegal. `T` is replaced by its bounds, which would likely be `Object`, and then the cast (generated by type-erasure) to `T[]` would fail because `Object[]` *IS-NOT-A* `T[]`. Figure 4.34 shows a generic version of `SimpleArrayList`, previously seen in Figure 4.23. The only tricky part is the code at line 38. Because we cannot create arrays of generic objects, we must create an array of `Object`, and then use a typecast. This typecast will generate a compiler warning about an unchecked type conversion. It is impossible to implement the generic collection classes with arrays without getting this warning. If clients want their code to compile without warnings, they should use only generic collection types, and not generic array types.

Arrays of Parameterized Types

Instantiation of arrays of parameterized types is illegal.

Instantiation of arrays of parameterized types is illegal. Consider the following code:

```
List<String> [ ] arr1 = new List<String>[ 10 ];
Object [ ] arr2 = arr1;
arr2[ 0 ] = new List<Double>( );
```

Normally, we would expect that the assignment at line 3, which has the wrong type, would generate an `ArrayStoreException`. However, after type erasure, the array type is `List[]`, and the object added to the array is `List`, so there is no `ArrayStoreException`. Thus, this code has no casts, yet will eventually generate a `ClassCastException`, which is exactly the situation that generics are supposed to avoid.

```
1  /**
2   * The GenericSimpleArrayList implements a growable array.
3   * Insertions are always done at the end.
4   */
5  public class GenericSimpleArrayList<AnyType>
6  {
7      /**
8       * Returns the number of items in this collection.
9       * @return the number of items in this collection.
10      */
11     public int size( )
12     {
13         return theSize;
14     }
15
16     /**
17      * Returns the item at position idx.
18      * @param idx the index to search in.
19      * @throws ArrayIndexOutOfBoundsException if index is bad.
20      */
21     public AnyType get( int idx )
22     {
23         if( idx < 0 || idx >= size( ) )
24             throw new ArrayIndexOutOfBoundsException( );
25         return theItems[ idx ];
26     }
27
28     /**
29      * Adds an item to this collection, at the end.
30      * @param x any object.
31      * @return true.
32      */
33     public boolean add( AnyType x )
34     {
35         if( theItems.length == size( ) )
36         {
37             AnyType [ ] old = theItems;
38               theItems = (AnyType [])new Object[size( )*2 + 1];
39             for( int i = 0; i < size( ); i++ )
40                 theItems[ i ] = old[ i ];
41         }
42
43         theItems[ theSize++ ] = x;
44         return true;
45     }
46
47     private static final int INIT_CAPACITY = 10;
48
49     private int theSize;
50     private AnyType [ ] theItems;
51 }
```

Figure 4.35 `SimpleArrayList` class using generics