1.5 Implementing Generic Components Using Java 5 Generics

Java 5 supports generic classes that are very 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.

1.5.1 Simple Generic Classes and Interfaces

Figure 1.9 shows a generic version of the MemoryCell class previously depicted in Figure 1.5. Here, we have changed the name to GenericMemoryCell because neither class is in a package and thus the names cannot be the same.

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 explicit restrictions on the type parameter, so the user can create types such as GenericMemoryCell<String> and 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. For example, in line 5 of Figure 1.9, the write method for GenericMemoryCell<String> requires a parameter of type String. Passing anything else will generate a compiler error.

Interfaces can also be declared as generic. For example, prior to Java 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 5, the Comparable class is generic, as shown in Figure 1.10. 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.

```
1 public class GenericMemoryCell<AnyType>
2 {
3     public AnyType read() {
4         return storedValue;
5     }
6     public void write(AnyType x) {
7         storedValue = x;
8     }
9     private AnyType storedValue;
10 }
```

Figure 1.9 Generic implementation of the MemoryCell class
1.5 Implementing Generic Components Using Java 5 Generics

Figure 1.10 Comparable interface, Java 5 version which is generic

1.5.2 Autoboxing/Unboxing

The code in Figure 1.7 is annoying to write because using the wrapper class requires creation of an Integer object prior to the call to add, and then the extraction of the int value from the Integer, using the intValue method. Prior to Java 5, this is required because if an int is passed in a place where an Integer object is required, the compiler will generate an error message, and if the result of an Integer object is assigned to an int, the compiler will generate an error message. This resulting code in Figure 1.7 accurately reflects the distinction between primitive types and reference types, yet it does not cleanly express the programmer's intent of storing ints in the collection.

Java 5 rectifies this situation. If an int is passed in a place where an Integer is required, the compiler will insert a call to the integer constructor behind the scenes. This is known as autoboxing. And if an Integer is passed in a place where an int is required, the compiler will insert a call to the intValue method behind the scenes. This is known as unboxing. Similar behavior occurs for the seven other primitive/wrapper pairs. Figure 1.11 illustrates the use of autoboxing and unboxing. Note that the entities referenced in the GenericMemoryCell are still Integer objects; int cannot be substituted for Integer in the GenericMemoryCell instantiations.

Figure 1.11 Autoboxing and unboxing
1.5.3 Wildcards with Bounds

Figure 1.12 shows a static method that computes the total area in an array of Shapes (we assume Shape is a class with an area method; Circle and Square extend Shape). Suppose we want to rewrite the method so that it works with a parameter that is Collection<Shape>. Collection is described in Chapter 3; for now, the only important thing about it is that it stores a collection of items that can be accessed with an enhanced for loop. Because of the enhanced for loop, the code should be identical, and the resulting code is shown in Figure 1.13. If we pass a Collection<Shape>, the code works. However, what happens if we pass a Collection<Square>? The answer depends on whether a Collection<Square> IS-A Collection<Shape>. Recall from Section 1.4.4 that the technical term for this is whether we have covariance.

In Java, as we mentioned in Section 1.4.4, arrays are covariant. So Square[] IS-A Shape[]. On the other hand, consistency would suggest that if arrays are covariant, then collections should be covariant too. On the other hand, as we saw in Section 1.4.4, 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, we cannot use covariance for collections.

What we are (which makes sense given that Java 5 makes superclasses of interfaces or even classes of types) is to write a totalArea method that does not work if passed a Collection<Square>

1.5.4 Gen

In some sense, there is a type hierarchy for interfaces, but there is no universal type hierarchy. Some reasons apply:

1. The type is object
2. The type is any
3. The type is a collection

If so, then an exercise search for value that uses Object Apple in an array.
1.5 Implementing Generic Components Using Java 5 Generics

```java
public static double totalArea( Collection<? extends Shape> arr )
{
    double total = 0;
    for( Shape s : arr )
        if( s != null )
            total += s.area();
    return total;
}
```

Figure 1.14 totalArea method revised with wildcards that works if passed a Collection<Square>

errors rather than runtime exceptions for type mismatches, generic collections are not
covariant. As a result, we cannot pass a Collection<Square> as a parameter to the method
in Figure 1.13.

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.

Java 5 makes up for this with wildcards. Wildcards are used to express subclasses (or
superclasses) of parameter types. Figure 1.14 illustrates the use of wildcards with a bound to
write a totalArea method that takes as parameter a Collection<T>, where T IS-A Shape. Thus,
Collection<Shape> and Collection<Square> are both 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); there are also some other
syntax uses that we do not discuss here.

1.5.4 Generic Static Methods

In some sense, the totalArea method in Figure 1.14 is generic, since it works for different
types. But there is no specific type parameter list, as was done in the GenericArray MemoryCell
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 1.15 illustrates a generic static method that performs a sequential
search for value x in array arr. By using a generic method instead of a nongeneric method
that uses Object as the parameter types, we can get compile-time errors if searching for an
Apple in an array of Shapes.
public static <AnyType> boolean contains( AnyType [] arr, AnyType x )
{
    for( AnyType val : arr )
        if( x.equals( val ) )
            return true;
    return false;
}

Figure 1.15  Generic static method to search an array

public static <AnyType> AnyType findMax( AnyType [] arr )
{
    int maxIndex = 0;
    for( int i = 1; i < arr.length; i++ )
        if( arr[ i ].compareTo( arr[ maxIndex ] ) > 0 )
            maxIndex = i;
    return arr[ maxIndex ];
}

Figure 1.16  Generic static method to find largest element in an array that does not work

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 precede the return type.

1.5.5 Type Bounds

Suppose we want to write a findMax routine. Consider the code in Figure 1.16. This code cannot work because the compiler cannot prove that the call to compareTo at line 6 is valid; 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 it 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

1.5.6 Type Erasure

Generic types, for the JVM Machine. Generic class known as type erasure generates a raw class removed. The type generic methods that class is used without.

One important different than the code not any faster. The stuff in the code, and the
1.5 Implementing Generic Components Using Java 5 Generics

```java
public static <AnyType extends Comparable<? super AnyType>>
AnyType findMax( AnyType[] arr )
{
    int maxIndex = 0;
    for( int i = 1; i < arr.length; i++ )
        if( arr[i].compareTo( arr[maxIndex] ) > 0 )
            maxIndex = i;
    return arr[ maxIndex ];
}
```

**Figure 1.17** Generic static method to find largest element in an array. Illustrates a bounds on the type parameter

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

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

Figure 1.17 shows the implementation of findMax. The compiler will accept arrays of types T only such that T implements the Comparable<T> 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.

1.5.6 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 nongeneric 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.

One important consequence of type erasure is that the generated code is not much different than the code that programmers have been writing before 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.
1.5.7 Restrictions on Generics

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

**Primitive Types**

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

**instanceof tests**

`instanceof` tests and typecasts work only with raw type. In the following code:

```java
GenericMemoryCell<Integer> cell1 = new GenericMemoryCell<Integer>();
   cell1.write(4);  
Object cell = cell1;
GenericMemoryCell<String> cell2 = (GenericMemoryCell<String>) cell;
String s = cell2.read();
```

the typecast succeeds at runtime since all types are `GenericMemoryCell`. Eventually, a runtime error results at the last line because the call to `read` tries to return a `String` but cannot. As a result, the typecast will generate a warning, and a corresponding `instanceof` test is illegal.

**Static Contexts**

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

```java
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 an array of a generic type. If `T` is a type variable, the statement

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

is illegal. `T` would be replaced by its bounds, which would probably be `Object`, and then the cast (generated by type erasure) to `T[]` would fail because `Object[]` is NOT A `T[]`. Because we cannot create arrays of generic objects, generally we must create an array of the erased type and then use a typecast. This typecast will generate a compiler warning about an unchecked type conversion.

**Arrays of Parameterized Types**

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

```java
1  GenericMem
2  GenericMem
3  Object []
4  arr2[0]
5  String s =

Normally, we would generate

GenericMemoryCell no ArrayStoreEx
ClassCastException
Avoid.
```

1.6 Fun

In Section 1.5, we method in Figure 1.17 that implement
decisions. In many cases, we assume that a method that it has
and a 5-by-5 rect
we are using area
an opening, the second example,
of strings, the do
decided to provide
The solution of objects and at the larger and will themselves; instan
An ingenious
both data and on
an instance of the
This object is cor
Figure 1.18 that takes a second pa
specified in java.
Any class that named compare in the following the sam
at line 9 can be a signal that if we a how to compare i
restrictions listed here

1.6 Function Objects

Normally, we would expect that the assignment at line 4, which has the wrong type, would generate an ArrayStoreException. However, after type erasure, the array type is GenericMemoryCell[], and the object added to the array is GenericMemoryCell, so there is no ArrayStoreException. Thus, this code has no casts, yet it will eventually generate a ClassCastException at line 5, which is exactly the situation that generics are supposed to avoid.

1.6 Function Objects

In Section 1.5, we showed how to write generic algorithms. As an example, the generic method in Figure 1.6 can be used to find the maximum item in an array.

However, that generic method has an important limitation: It works only for objects that implement the Comparable interface, using compareTo as the basis for all comparison decisions. In many situations, this approach is not feasible. For instance, it is a stretch to presume that a Rectangle class will implement Comparable, and even if it does, the compareTo method that it has might not be the one we want. For instance, given a 2-by-10 rectangle and a 5-by-5 rectangle, which is the larger rectangle? The answer would depend on whether we are using area or width to decide. Or perhaps if we are trying to fit the rectangle through an opening, the larger rectangle is the rectangle with the larger minimum dimension. As a second example, if we wanted to find the maximum string (alphabetically last) in an array of strings, the default compareTo does not ignore case distinctions, so "ZEBRA" would be considered to precede "alligator" alphabetically, which is probably not what we want.

The solution in these situations is to rewrite findMax to accept two parameters: an array of objects and a comparison function that explains how to decide which of two objects is the larger and which is the smaller. In effect, the objects no longer know how to compare themselves; instead, this information is completely decoupled from the objects in the array.

An ingenious way to pass functions as parameters is to notice that an object contains both data and methods, so we can define a class with no data and one method, and pass an instance of the class. In effect, a function is being passed by placing it inside an object. This object is commonly known as a function object.

Figure 1.18 shows the simplest implementation of the function object idea. findMax takes a second parameter, which is an object of type Comparator. The Comparator interface is specified in java.util and contains a compare method. This interface is shown in Figure 1.19.

Any class that implements the Comparator<AnyType> interface type must have a method named compare that takes two parameters of the generic type (AnyType) and returns an int, following the same general contract as compareTo. Thus, in Figure 1.18, the call to compare at line 9 can be used to compare array items. The bounded wildcard at line 4 is used to signal that if we are finding the maximum in an array of items, the comparator must know how to compare items, or objects of the items' supertype. To use this version of findMax, at

```java
GenericMemoryCell<String> [] arr1 = new GenericMemoryCell<String>[10];
GenericMemoryCell<Double> cell = new GenericMemoryCell<Double>() { cell.write(4.5); };
Object [] arr2 = arr1;
arr2[0] = cell;
String s = arr1[0].read();
```