Generics

Introduced by JDK 5, generics changed Java in two important ways. First, it added a new syntactical element to the language. Second, it caused changes to many of the classes and methods in the core API.

Through the use of **generics**, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data.

Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type **Integer**, **String**, **Object**, or **Thread**.

With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort.

What Are Generics?

At its core, the term *generics* means *parameterized types.* Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

Using *generics*, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.*

Generics add the type safety that was lacking in **Object**. They also streamline the process, because it is no longer necessary to explicitly employ casts to translate between **Object** and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit.
Thus, generics expand your ability to reuse code and let you do so safely and easily.

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.
```

```
class Gen<T> {
    T ob;      // declare an object of type T
 // Pass the constructor a reference to an object of type
T.
    Gen(T o) {
          ob = o;
          }     // Return ob.
    T getob() {
              return ob;
          }
// Show type of T.
    void showType() {
        System.out.println("Type of T is " +
                ob.getClass().getName());
        }
}
```

```
// Demonstrate the generic class.
class GenDemo {
  public static void main(String args[]) {
    // Create a Gen reference for Integers.
   Gen<Integer> iOb;
    // Create a Gen<Integer> object and assign its
    // reference to iOb. Notice the use of autoboxing
    // to encapsulate the value 88 within an Integer
object.
    iOb = new Gen<Integer>(88);
    // Show the type of data used by iOb.
    iOb.showType();
    // Get the value in iOb. Notice that no cast is needed.
    int v = iOb.getob();
    System.out.println("value: " + v);
    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");
    // Show the type of data used by strOb.
    strOb.showType();
    // Get the value of strOb. Notice that no cast is needed.
    String str = strOb.getob();
    System.out.println("value: " + str);
    }
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

Generics Work Only with Objects

When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type. You cannot use a primitive type, such as **int** or **char**. Therefore, the following declaration is illegal:

Gen<int> strOb = new Gen<int>(53); // Error, can't use primitive type

Generic Types Differ Based on Their Type Arguments

A reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

iOb = strOb; // Wrong!

Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type parameters differ. This is part of the way that generics add type safety and prevent errors.

How Generics Improve Type Safety

Generics automatically ensure the type safety of all operations involving **Gen**. In the process, they eliminate the need for you to enter casts and to type-check code by hand. The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using **Object** references to create "generic" code has always been possible, that code was not type safe, and its misuse could result in run-time exceptions. Generics prevent this from occurring. In essence, through generics, what were once run-time errors have become compile-time errors. This is a major advantage.

To specify two or more type parameters, simply use a comma-separated list.

```
// A generic class with two type parameters: T and V.
class TwoGen<T, V> {
   T ob1;
   V ob2;
// Pass the constructor a reference
// to an object of type T and an object of type V.
  TwoGen(T o1, V o2) {
  ob1 = o1;        ob2 = o2;
   }
// Show types of T and V.
  void showTypes() {
     System.out.println("Type of T is " +
                                        ob1.getClass().getName());
   System.out.println("Type of V is " +
                                             ob2.getClass().getName());
}
 T getob1() {
        return ob1;
     }
     V getob2() {
        return ob2;
      }
}
// Demonstrate TwoGen.
class SimpGen {
   public static void main(String args[]) {
      TwoGen<Integer, String> tgObj =
              new TwoGen<Integer, String>(88, "Generics");
```

```
// Show the types.
    tgObj.showTypes();
// Obtain and show values.
    int v = tgObj.getob1();
    System.out.println("value: " + v);

    String str = tgObj.getob2();
    System.out.println("value: " + str);
  }
}
```

The output from this program is shown here:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized.

Here is the syntax for declaring a generic class:

class *class-name*<*type-param-list*> { // ...

Here is the syntax for declaring a reference to a generic class:

class-name<*type-arg-list*> *var-name* =
                    new *class-name*<*type-arg-list*>(*cons-arg-list*);

Bounded Types

Sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles. Thus, you want to specify type of the numbers generically, using a type parameter. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an **extends** clause when specifying the type parameter, as shown here:

<*T* extends *superclass*>

This specifies that *T* can only be replaced by *superclass,* or subclasses of *superclass.* Thus, *superclass* defines an inclusive, upper limit.

You can use an upper bound to fix the **Stats** class specifying **Number** as an upper bound, as shown here:

```
// In this version of Stats, the type argument for  T must
//be either Number, or a class derived from Number.
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass
     // Pass the constructor a reference to
     // an array of type Number or subclass.
     Stats(T[] o) {
        nums = o;
     }
    // Return type double in all cases.
    double average() {
    double sum = 0.0;
    for(int i=0; i < nums.length; i++)
    sum += nums[i].doubleValue();
    return sum / nums.length;
   }
}
// Demonstrate Stats.
class BoundsDemo {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + v);
```

```
        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);
      // This won't compile because String is not a
      // subclass of Number.
      // String strs[] = { "1", "2", "3", "4", "5" };
      // Stats<String> strob = new Stats<String>(strs);
      // double x = strob.average();
     // System.out.println("strob average is " + v);
    }
}
```

The output is shown here:

```
Average is 3.0
Average is 3.3
```

The bounding of **T** also prevents nonnumeric **Stats** objects from being created. For example, if you try removing the comments from the lines at the end of the program, and then try recompiling, you will receive compile-time errors because **String** is not a subclass of **Number**. In addition to using a class type as a bound, you can also use an/multiple interface type. A bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal. Use the **&** operator to connect them. For example,
class Gen<T extends MyClass & MyInterface> { // ...
Here, **T** is bounded by a class called **MyClass** and an interface called **MyInterface**. Thus, any type argument passed to **T** must be a subclass of **MyClass** and implement **MyInterface**.

Using Wildcard Arguments

As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs. For example, given the **Stats** class shown at the end of the preceding section, assume that you want to add a method called **sameAvg( )** that determines if two **Stats** objects contain arrays that yield the same average, no matter what type of numeric data each object holds. For example, if one object contains the **double** values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same. One way to implement **sameAvg( )** is to pass it a **Stats** argument, and then compare the average of that argument against the invoking object, returning true only if the averages are the same. For example, you want to be able to call **sameAvg( )**, as shown here:

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);
if(iob.sameAvg(dob))
System.out.println("Averages are the same.");
else
System.out.println("Averages differ.");
```

At first, creating **sameAvg( )** seems like an easy problem. Because **Stats** is generic and its **average( )** method can work on any type of **Stats** object, it seems that creating **sameAvg( )** would be straightforward. Unfortunately, trouble starts as soon as you try to declare a parameter of type **Stats**. Because **Stats** is a parameterized type, what do you specify for **Stats**’ type parameter when you declare a parameter of that type?

you might think of a solution like this, in which **T** is used as the type parameter:

```
// This won't work!
// Determine if two averages are the same.
boolean sameAvg(Stats<T> ob) {
if(average() == ob.average())
return true;
return false;
}
```

The trouble with this attempt is that it will work only with other **Stats** objects whose type is the same as the invoking object. For example, if the invoking object is of type **Stats<Integer>**, then the parameter **ob** must also be of type **Stats<Integer>**. It can't be used to compare the average of an object of type **Stats<Double>** with the average of an object of type **Stats<Short>**, for example.

you must use another feature of Java generics: the *wildcard* argument. The wildcard argument is specified by the **?**, and it represents an unknown type. Using a wildcard, here is one way to write the **sameAvg( )** method:

```
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob) {
if(average() == ob.average())
return true;
return false;
}
```

Here, **Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their averages compared. The following program demonstrates this:

```
// Use a wildcard.
class Stats<T extends Number> {
     T[] nums; // array of Number or subclass
     // Pass the constructor a reference to
     // an array of type Number or subclass.
     Stats(T[] o) {
     nums = o;
     }
   // Return type double in all cases.
   double average() {
     double sum = 0.0;
     for(int i=0; i < nums.length; i++)
          sum += nums[i].doubleValue();
          return sum / nums.length;
     }
// Determine if two averages are the same.
// Notice the use of the wildcard.
```

```
boolean sameAvg(Stats<?> ob) {
if(average() == ob.average())
return true;
return false;
}
}
// Demonstrate wildcard.
class WildcardDemo {
public static void main(String args[]) {
Integer inums[] = { 1, 2, 3, 4, 5 };
Stats<Integer> iob = new Stats<Integer>(inums);
double v = iob.average();
System.out.println("iob average is " + v);
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Double> dob = new Stats<Double>(dnums);
double w = dob.average();
System.out.println("dob average is " + w);
Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
Stats<Float> fob = new Stats<Float>(fnums);
double x = fob.average();
System.out.println("fob average is " + x);
// See which arrays have same average.
System.out.print("Averages of iob and dob ");
if(iob.sameAvg(dob))
System.out.println("are the same.");
else
System.out.println("differ.");
System.out.print("Averages of iob and fob ");
if(iob.sameAvg(fob))
System.out.println("are the same.");
else
System.out.println("differ.");
}
}
```

The output is shown here:

```
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

One last point: It is important to understand that the wildcard does not affect what type of **Stats** objects can be created. This is governed by the **extends** clause in the **Stats** declaration. The wildcard simply matches any *valid* **Stats** object.

Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy. To understand why, let's work through an example.

Consider the following hierarchy of classes that encapsulate coordinates:

```
// Two-dimensional coordinates.
class TwoD {
    int x, y;
    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
// Three-dimensional coordinates.
class ThreeD extends TwoD {
	int z;
	ThreeD(int a, int b, int c) {
		super(a, b);
		z = c;
	}
}
// Four-dimensional coordinates.
class FourD extends ThreeD {
	int t;
	FourD(int a, int b, int c, int d) {
		super(a, b, c);
		t = d;
	}
}
```

At the top of the hierarchy is **TwoD**, which encapsulates a two-dimensional, XY coordinate.
TwoD is inherited by **ThreeD**, which adds a third dimension, creating an XYZ coordinate.
ThreeD is inherited by **FourD**, which adds a fourth dimension (time), yielding a four-dimensional coordinate.
Shown next is a generic class called **Coords**, which stores an array of coordinates:

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
      T[] coords;
      Coords(T[] o)   { coords = o;    }
}
```

Notice that **Coords** specifies a type parameter bounded by **TwoD**. This means that any array stored in a **Coords** object will contain objects of type **TwoD** or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the **coords** array of a **Coords** object. Because all types of **Coords** objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown here:

```
static void showXY(Coords<?> c) {
     System.out.println("X Y Coordinates:");
     for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " + c.coords[i].y);
          System.out.println();
  }
```

Because **Coords** is a bounded generic type that specifies **TwoD** as an upper bound, all objects that can be used to create a **Coords** object will be arrays of type **TwoD**, or of classes derived from **TwoD**. Thus, **showXY( )** can display the contents of any **Coords** object.

However, what if you want to create a method that displays the X, Y, and Z coordinates of a **ThreeD** or **FourD** object? The trouble is that not all **Coords** objects will have three coordinates, because a **Coords<TwoD>** object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for **Coords<ThreeD>** and **Coords<FourD>** objects, while preventing that method from being used with **Coords<TwoD>** objects? The answer is the *bounded wildcard argument.*

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate.

The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a **Coords** object, if that object actually has those three coordinates.

For example, the following **showXYZ( )** method shows the X, Y, and Z coordinates of the elements stored in a **Coords** object, if those elements are actually of type **ThreeD** (or are derived from **ThreeD)**:

```
static void showXYZ(Coords<? extends ThreeD> c) {
System.out.println("X Y Z Coordinates:");
for(int i=0; i < c.coords.length; i++)
System.out.println(c.coords[i].x + " " +
c.coords[i].y + " " +
c.coords[i].z);
System.out.println();
}
```

Notice that an **extends** clause has been added to the wildcard in the declaration of parameter **c**. It states that the **?** can match any type as long as it is **ThreeD**, or a class derived from **ThreeD**. Thus, the **extends** clause establishes an upper bound that the **?** Can match. Because of this bound, **showXYZ( )** can be called with references to objects of type **Coords<ThreeD>** or **Coords<FourD>**, but not with a reference of type **Coords<TwoD>**.

Attempting to call **showXZY( )** with a **Coords<TwoD>** reference results in a compile-time error, thus ensuring type safety.
Here is an entire program that demonstrates the actions of a bounded wildcard argument:

```
// Bounded Wildcard arguments.
// Two-dimensional coordinates.
class TwoD {
	int x, y;
	TwoD(int a, int b) {
		x = a;
		y = b;
	}
}
// Three-dimensional coordinates.
class ThreeD extends TwoD {
	int z;
	ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
        }
}
// Four-dimensional coordinates.
class FourD extends ThreeD {
        int t;
        FourD(int a, int b, int c, int d) {
                super(a, b, c);
                t = d;
        }
}
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
        T[] coords;
        Coords(T[] o)  { coords = o; }
}
// Demonstrate a bounded wildcard.
```

```
class BoundedWildcard {
          static void showXY(Coords<?> c) {
                    System.out.println("X Y Coordinates:");
                    for(int i=0; i < c.coords.length; i++)
                    System.out.println(c.coords[i].x + " " +c.coords[i].y);
                    System.out.println();           }
         static void showXYZ(Coords<? extends ThreeD> c) {
          System.out.println("X Y Z Coordinates:");
          for(int i=0; i < c.coords.length; i++)
          System.out.println(c.coords[i].x + " " +c.coords[i].y + " " +
                                                  c.coords[i].z);
          System.out.println();           }
        static void showAll(Coords<? extends FourD> c) {
          System.out.println("X Y Z T Coordinates:");
          for(int i=0; i < c.coords.length; i++)
          System.out.println(c.coords[i].x + " " + c.coords[i].y + " " +
                              c.coords[i].z + " " + c.coords[i].t);
          System.out.println();   }
public static void main(String args[]) {
    TwoD td[] = { new TwoD(0, 0), new TwoD(7, 9),
                              new TwoD(18, 4), new TwoD(-1, -23) };
    Coords<TwoD> tdlocs = new Coords<TwoD>(td);
    System.out.println("Contents of tdlocs.");
    showXY(tdlocs); // OK, is a TwoD
    // showXYZ(tdlocs); // Error, not a ThreeD
    // showAll(tdlocs); // Error, not a FourD
    // Now, create some FourD objects.
    FourD fd[] = { new FourD(1, 2, 3, 4), new FourD(6, 8, 14, 8),
            new FourD(22, 9, 4, 9), new FourD(3, -2, -23, 17) };
    Coords<FourD> fdlocs = new Coords<FourD>(fd);
    System.out.println("Contents of fdlocs.");
    // These are all OK.
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
    }    }
```

The output from the program is shown here:

```
Contents of tdlocs.
X Y Coordinates:
0 0
7 9
18 4
-1 -23
Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2
X Y Z Coordinates:
1 2 3
6 8 14
22 9 4
3 -2 -23
```

```
X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

Notice these commented-out lines:

```
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
```

Because **tdlocs** is a **Coords(TwoD)** object, it cannot be used to call **showXYZ( )** or **showAll( )** because bounded wildcard arguments in their declarations prevent it. To prove this to yourself, try removing the comment symbols, and then attempt to compile the program. You will receive compilation errors because of the type mismatches.

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

<? extends *superclass*>

where *superclass* is the name of the class that serves as the upper bound. Remember, this is an inclusive clause because the class forming the upper bound (that is, specified by *superclass*) is also within bounds.
You can also specify a lower bound for a wildcard by adding a **super** clause to a wildcard declaration. Here is its general form:
<? super *subclass*>
In this case, only classes that are superclasses of *subclass* are acceptable arguments. This is an exclusive clause, because it will not match the class specified by *subclass.*
Creating a Generic Method
it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.
Let's begin with an example. The following program declares a non-generic class called **GenMethDemo** and a static generic method within that class called **isIn( )**. The **isIn( )** method determines if an object is a member of an array.

It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```
// Demonstrate a simple generic method.
class GenMethDemo {
    // Determine if an object is in an array.
    static <T, V extends T> boolean isIn(T x, V[] y) {
        for(int i=0; i < y.length; i++)
            if(x.equals(y[i])) return true;
        return false;
    }
    public static void main(String args[]) {
        // Use isIn() on Integers.
        Integer nums[] = { 1, 2, 3, 4, 5 };
        if(isIn(2, nums))
            System.out.println("2 is in nums");
        if(!isIn(7, nums))
            System.out.println("7 is not in nums");
        System.out.println();
        // Use isIn() on Strings.
        String strs[] = { "one", "two", "three", "four", "five" };
        if(isIn("two", strs))
                System.out.println("two is in strs");
        if(!isIn("seven", strs))
                System.out.println("seven is not in strs");
        // Oops! Won't compile! Types must be compatible.
        // if(isIn("two", nums))
                // System.out.println("two is in strs");
        }
}
```

The output from the program is shown here:

```
2 is in nums
7 is not in nums
two is in strs
seven is not in strs
```

This ability to enforce type safety is one of the most important advantages of generic methods.

The syntax used to create **isIn( )** can be generalized. Here is the syntax for a generic method:

<*type-param-list*> *ret-type meth-name*(*param-list*) { // ...

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Generic Constructors

It is also possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```
// Use a generic constructor.
class GenCons {
      private double val;
      <T extends Number> GenCons(T arg) {
            val = arg.doubleValue();
          }
      void showval()    {  System.out.println("val: " + val);  }
}
```

```
class GenConsDemo {
    public static void main(String args[]) {
    GenCons  test = new GenCons(100);
    GenCons  test2 = new GenCons(123.5F);
    test.showval();
    test2.showval();
    }
}
```

The output is shown here:

```
val: 100.0
val: 123.5
```

Because **GenCons( )** specifies a parameter of a generic type, which must be a subclass of **Number**, **GenCons( )** can be called with any numeric type, including **Integer**, **Float**, or **Double**. Therefore, even though **GenCons** is not a generic class, its constructor is generic.

Generic Interfaces

Generic interfaces are specified just like generic classes. Here is an example. It creates an interface called **MinMax** that declares the methods **min( )** and **max( )**, which are expected to return the minimum and maximum value of some set of objects.

```
// A generic interface example.
// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
T min();
T max();
}
// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements
MinMax<T> {
T[] vals;
MyClass(T[] o) { vals = o; }
// Return the minimum value in vals.
```

```
    public T min() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) < 0) v = vals[i];
        return v;
    }
    // Return the maximum value in vals.
    public T max() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) > 0) v = vals[i];
        return v;
    }
}
class GenIFDemo {
    public static void main(String args[]) {
        Integer inums[] = {3, 6, 2, 8, 6 };
        Character chs[] = {'b', 'r', 'p', 'w' };
MyClass<Integer> iob = new MyClass<Integer>(inums);
MyClass<Character> cob = new MyClass<Character>(chs);
System.out.println("Max value in inums: " + iob.max());
System.out.println("Min value in inums: " + iob.min());
System.out.println("Max value in chs: " + cob.max());
System.out.println("Min value in chs: " + cob.min());
}
}
```

The output is shown here:

```
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b
```

a generic interface is declared in the same way as is a generic class. In this case, the type parameter is **T**, and its upper bound is **Comparable**, which is an interface defined by **java.lang**. A class that implements **Comparable** defines objects that can be ordered.

Thus, requiring an upper bound of **Comparable** ensures that **MinMax** can be used only with objects that are capable of being compared.

```
class MyClass<T extends Comparable<T>> implements
MinMax<T> {
```

Pay special attention to the way that the type parameter **T** is declared by **MyClass** and then passed to **MinMax**. Because **MinMax** requires a type that implements **Comparable**, the implementing class (**MyClass** in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the **implements** clause. In fact, it would be wrong to do so.

```
// This is wrong!
class MyClass<T extends Comparable<T>>
    implements MinMax<T extends Comparable<T>> {
```

In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface.

For example, the following attempt to declare **MyClass** is in error:
class MyClass implements MinMax<T> { // Wrong!
if a class implements a *specific type* of generic interface, such as shown here:
class MyClass implements MinMax<Integer> { // OK
then the implementing class does not need to be generic.
The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented.
Here is the generalized syntax for a generic interface:
interface *interface-name*<*type-param-list*> { // ...
Here, *type-param-list* is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:
class *class-name*<*type-param-list*>
implements *interface-name*<*type-arg-list*> {

Raw Types and Legacy Code

Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generic code.

To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a *raw type* for the class. This raw type is compatible with legacy code, which has no knowledge of generics. The main drawback to using the raw type is that the type safety of generics is lost.

Here is an example that shows a raw type in action:

```
// Demonstrate a raw type.
class Gen<T> {
    T ob; // declare an object of type T
    // Pass the constructor a reference to an object of type T.
    Gen(T o)    { ob = o;  }
    // Return ob.
    T getob()   {  return ob;  }
}
```

```
// Demonstrate raw type.
class RawDemo {
    public static void main(String args[]) {
    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);
    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");
    // Create a raw-type Gen object and give it a Double value.
    Gen raw = new Gen(new Double(98.6));
    // Cast here is necessary because type is unknown.
    double d = (Double) raw.getob();
    System.out.println("value: " + d);
    // The use of a raw type can lead to run-time  exceptions.         //
some examples.  following cast causes a run-time error!
    // int i = (Integer) raw.getob(); // run-time error
    // This assignment overrides type safety.
    strOb = raw;     // OK, but potentially wrong
    // String str = strOb.getob();     // run-time error
    // This assignment also overrides type safety.
    raw = iOb; // OK, but potentially wrong
    // d = (Double) raw.getob(); // run-time error
    }
}
```
This program contains several interesting things. First, a raw type of the generic **Gen** class is created by the following declaration:

```
Gen raw = new Gen(new Double(98.6));
```
A raw type is not type safe. Thus, a variable of a raw type can be assigned a reference to any type of **Gen** object. The reverse is also allowed; a variable of a specific **Gen** type can be assigned a reference to a raw **Gen** object. However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented.

Because of the potential for danger inherent in raw types, **javac** displays *unchecked warnings* when a raw type is used in a way that might jeopardize type safety.

In the preceding program, these lines generate unchecked warnings:

```
Gen raw = new Gen(new Double(98.6));
strOb = raw; // OK, but potentially wrong
```

At first, you might think that this line should also generate an unchecked warning, but it does not:

```
raw = iOb; // OK, but potentially wrong
```

No compiler warning is issued because the assignment does not cause any *further* loss of type safety than had already occurred when **raw** was created.

One final point: You should limit the use of raw types to those cases in which you must mix legacy code with newer, generic code. Raw types are simply a transitional feature and not something that should be used for new code.

Erasure
An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of *erasure.*
In general, here is how erasure works. When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is **Object** if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Ambiguity Errors

The inclusion of generics gives rise to a new type of error that you must guard against: *ambiguity.* Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on overloaded methods.
class MyGenClass<T, V> {
    T ob1;
    V ob2;
// These two overloaded methods are ambiguous and not compile.
    void set(T o) {
        ob1 = o;
    }
    void set(V o) {
        ob2 = o;
    }
}
```

Notice that **MyGenClass** declares two generic types: **T** and **V**. Inside **MyGenClass**, an attempt is made to overload **set( )** based on parameters of type **T** and **V**. This looks reasonable because **T** and **V** appear to be different types. However, there are two ambiguity problems here.

First, as **MyGenClass** is written, there is no requirement that **T** and **V** actually be different types. For example, it is perfectly correct (in principle) to construct a **MyGenClass** object as shown here:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

In this case, both **T** and **V** will be replaced by **String**. This makes both versions of **set( )** identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of **set( )** reduces both versions to the following:

```
void set(Object o) { // ...
```

Thus, the overloading of **set( )** as attempted in **MyGenClass** is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that **V** will always be some type of **String**, you might try to fix **MyGenClass** by rewriting its declaration as shown here:

```
class MyGenClass<T, V extends String> { // almost OK!
```

This change causes **MyGenClass** to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();
```

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

```
MyGenClass<String, String> x = new MyGenClass<String, String>();
```

In this case, since both **T** and **V** are **String**, which version of **set( )** is to be called?

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload **set( )**. Often, the solution to ambiguity involves the restructuring of the code, because ambiguity often means that you have a conceptual error in your design.

Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

Type Parameters Can’t Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
      T ob;
      Gen() {  ob = new T();  // Illegal!!!
      }
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to understand: because **T** does not exist at run time, how would the compiler know what type of object to create? Remember, erasure removes all type parameters during the compilation process.

Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example, all of the **static** members of this class are illegal:

```
class Wrong<T> {
    // Wrong, no static variables of type T.
    static T ob;
    // Wrong, no static method can use T.
    static T getob() {
        return ob;
      }
    // Wrong, no static method can access object
    // of type T.
    static void showob() {
        System.out.println(ob);
      }
}
```

Although you can’t declare **static** members that use a type parameter declared by the enclosing class, you *can* declare **static** generic methods, which define their own type parameters, as was done earlier in this chapter.

Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose base type is a type parameter. Second, you cannot create an array of typespecific generic references. The following short program shows both situations:

```
// Generics and arrays.
class Gen<T extends Number> {
    T ob;
    T vals[]; // OK
    Gen(T o, T[] nums) {
        ob = o;
    // This statement is illegal.
```

```
        // vals = new T[10]; // can't create an array of T
        // But, this statement is OK.
        vals = nums; // OK to assign reference to existent array
        }
}
class GenArrays {
        public static void main(String args[]) {
                Integer n[] = { 1, 2, 3, 4, 5 };
                Gen<Integer> iOb = new Gen<Integer>(50, n);
        // Can't create an array of type-specific generic references.
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
        // This is OK.
                Gen<?> gens[] = new Gen<?>[10]; // OK
        }
}
```

As the program shows, it’s valid to declare a reference to an array of type **T**, as this line does:

```
        T vals[]; // OK
```

But, you cannot instantiate an array of **T**, as this commented-out line attempts:
// vals = new T[10]; // can't create an array of T
The reason you can't create an array of **T** is that **T** does not exist at run time, so there is no way for the compiler to know what type of array to actually create.
However, you can pass a reference to a type-compatible array to **Gen( )** when an object is created and assign that reference to **vals**, as the program does in this line:
vals = nums; // OK to assign reference to existent array
This works because the array passed to **Gen** has a known type, which will be the same type as **T** at the time of object creation.
Inside **main( )**, notice that you can't declare an array of references to a specific generic type.
That is, this line won't compile.
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

Arrays of specific generic types simply aren’t allowed, because they can lead to a loss of type safety.

You *can* create an array of references to a generic type if you use a wildcard, however, as shown here:

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Exception Restriction

A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

Type Inference with Diamond Operator

class-name<*type-arg-list*> *var-name* =
new *class-name*<>(*cons-arg-list*);

Here new clause is empty. Type args need not be specified again.
This is referred to as *diamond* operator. (added in JDK 7). E.g.

```
TwoGen<Integer, String> tgObj =  new TwoGen<>(88, "Generics");
```

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a non-generic class. Thus, a generic class can act as a superclass or be a subclass. The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses. This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Superclass

Here is a simple example of a hierarchy that uses a generic superclass:

```
// A simple generic class hierarchy.
class Gen<T> {
    T ob;
    Gen(T o) {
        ob = o;
    }
    // Return ob.
    T getob() {
        return ob;
    }
}
// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
    V ob2;
    Gen2(T o, V o2) {
        super(o);
        ob2 = o2;
    }
    V getob2() {
        return ob2;
    }
}
// Create an object of type Gen2.
```

```
class HierDemo {
    public static void main(String args[]) {
        // Create a Gen2 object for String and Integer.
        Gen2<String, Integer> x =
        new Gen2<String, Integer>("Value is: ", 99);
        System.out.print(x.getob());
        System.out.println(x.getob2());
    }
}
```

Notice the declaration of this version of **Gen2**, which is shown here:

```
class Gen2<T, V> extends Gen<T> {
```

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2**, and as a return type for the method **getob2( )**. In **main( )**, a **Gen2** object is created in which type parameter **T** is **String**, and type parameter **V** is **Integer**. The program displays the following, expected, result:

```
Value is: 99
```