Introduction to generic types in JDK 5.0

**Introduction to generics**

**What are generics?**

Generic types, or *generics* for short, are an extension to the Java language's type system to support the creation of classes that can be parameterized by types. You can think of a type parameter as a placeholder for a type to be specified at the time the parameterized type is used, just as a formal method argument is a placeholder for a value that is passed at runtime.

The motivation for generics can be seen in the Collections framework. For example, the Map class allows you to add entries of any class to a Map, even though it is a very common use case to store objects of only a certain type, such as String, in a given map.

Because Map.get() is defined to return Object, you typically have to cast the results of Map.get() to the expected type, as in the following code:

|  |
| --- |
| Map m = new HashMap();  m.put("key", "blarg");  String s = (String) m.get("key"); |

To make the program compile, you have to cast the result of get() to String, and hope that the result really is a String. But it is possible that someone has stored something other than a String in this map, in which case the code above would throw aClassCastException.

Ideally, you would like to capture the concept that m is a Map that maps String keys to String values. This would let you eliminate the casts from your code and, at the same time, gain an additional layer of type checking that would prevent someone from storing keys or values of the wrong type in a collection. This is what generics do for you.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section2.html#ibm-pcon)

**Benefits of generics**

The addition of generics to the Java language is a major enhancement. Not only were there major changes to the language, type system, and compiler to support generic types, but the class libraries were overhauled so that many important classes, such as the Collections framework, have been made generic. This enables a number of benefits:

* **Type safety.** The primary goal of generics is to increase the type safety of Java programs. By knowing the type bounds of a variable that is defined using a generic type, the compiler can verify type assumptions to a much greater degree. Without generics , these assumptions exist only in the programmer's head (or if you are lucky, in a code comment).

A popular technique in Java programs is to define collections whose elements or keys are of a common type, such as "list of String" or "map from String to String." By capturing that additional type information in a variable's declaration, generics enable the compiler to enforce those additional type constraints. Type errors can now be caught at compile time, rather than showing up as ClassCastExceptions at runtime. Moving type checking from runtime to compile time helps you find errors more easily and improves your programs' reliability.

* **Elimination of casts.** A side benefit of generics is that you can eliminate many type casts from your source code. This makes code more readable and reduces the chance of error.

Although the reduced need for casting reduces the verbosity of code that uses generic classes, declaring variables of generic types involves a corresponding increase in verbosity. Compare the following two code examples.

This code does not use generics:

|  |
| --- |
| List li = new ArrayList();  li.put(new Integer(3));  Integer i = (Integer) li.get(0); |

This code uses generics:

|  |
| --- |
| List<Integer> li = new ArrayList<Integer>();  li.put(new Integer(3));  Integer i = li.get(0); |

Using a variable of generic type only once in a simple program does not result in a net savings in verbosity. But the savings start to add up for larger programs that use a variable of generic type many times.

* **Potential performance gains.** Generics create the possibility for greater optimization. In the initial implementation of generics, the compiler inserts the same casts into the generated bytecode that the programmer would have specified without generics. But the fact that more type information is available to the compiler allows for the possibility of optimizations in future versions of the JVM.

Because of the way generics are implemented, (almost) no JVM or classfile changes were required for the support of generic types. All of the work is done in the compiler, which generates code similar to what you would write without generics (complete with casts), only with greater confidence in its type safety.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section2.html#ibm-pcon)

**Example of generic usage**

Many of the best examples of generic types come from the Collections framework, because generics let you specify type constraints on the elements stored in collections. Consider this example of using the Map class, which involves a certain degree of optimism that the result returned by Map.get() really will be a String:

|  |
| --- |
| Map m = new HashMap();  m.put("key", "blarg");  String s = (String) m.get("key"); |

The above code will throw a ClassCastException in the event someone has placed something other than a String in the map. Generics allow you to express the type constraint that m is a Map that maps String keys to String values. This lets you eliminate the casts from your code and, at the same time, gain an additional layer of type checking that would prevent someone from storing keys or values of the wrong type in a collection.

The following code sample shows a portion of the definition of the Map interface from the Collections framework in JDK 5.0:

|  |
| --- |
| public interface Map<K, V> {  public void put(K key, V value);  public V get(K key);  } |

Note two additions to the interface:

* The specification of *type parameters* K and V at the class level, representing placeholders for types that will be specified when a variable of type Map is declared
* The use of K and V in the method signature for get(), put(), and other methods

To gain the benefit of using generics, you must supply concrete values for K and V when defining or instantiating variables of typeMap. You do this in a relatively straightforward way:

|  |
| --- |
| Map<String, String> m = new HashMap<String, String>();  m.put("key", "blarg");  String s = m.get("key"); |

When you use the generic version of Map, you no longer need to cast the result of Map.get() to String, because the compiler knows that get() will return a String.

You don't save any keystrokes in the version that uses generics; in fact, it requires more typing than the version that uses the cast. The savings comes in the form of the additional type safety you get by using generic types. Because the compiler knows more about the types of keys and values that will be put into a Map, type checking moves from execution time to compile time, improving reliability and speeding development.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section2.html#ibm-pcon)

**Backward compatibility**

An important goal for the addition of generics to the Java language was to maintain backward compatibility. Although many classes in the standard class libraries in JDK 5.0 have been generified, such as the Collections framework, existing code that uses Collections classes such as HashMap and ArrayList will continue to work unmodified in JDK 5.0. Of course, existing code that does not take advantage of generics will not gain the additional type-safety benefits of generics.

**Basics of generic types**

**Type parameters**

When you define a generic class, or declare a variable of a generic class, you use angle brackets to specify *formal type parameters*. The relationship between formal and actual type parameters is similar to the relationship between formal and actual method parameters, except that type parameters represent types, not values.

Type parameters in a generic class can be used almost anywhere a class name can be used. For example, here is an excerpt from the definition of the java.util.Map interface:

|  |
| --- |
| public interface Map<K, V> {  public void put(K key, V value);  public V get(K key);  } |

The Map interface is parameterized by two types -- the key type K and the value type V. Methods that would (without generics) accept or return Object now use K or V in their signatures instead, indicating additional typing constraints underlying the specification of Map.

When declaring or instantiating objects of a generic type, you must specify the values of the type parameters:

|  |
| --- |
| Map<String, String> map = new HashMap<String, String>(); |

Note that in this example, you have to specify the type parameters twice -- once in declaring the type of the variable map and a second time in selecting the parameterization of the HashMap class so you can instantiate an instance of the correct type.

When the compiler encounters a variable of type Map<String, String>, it knows that K and V now are bound to String, and so it knows that the result of Map.get() on such a variable will have type String.

Any class, except an exception type, an enumeration, or an anonymous inner class, can have type parameters.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#ibm-pcon)

**Naming type parameters**

The recommended naming convention is to use uppercase, single-letter names for type parameters. This differs from the C++ convention (see [Appendix A: Comparison to C++ templates](http://www.ibm.com/developerworks/java/tutorials/j-generics/section8.html#appendix)), and reflects the assumption that most generic classes will have a small number of type parameters. For common generic patterns, the recommended names are:

* K - A key, such as the key to a map
* V - A value, such as the contents of a List, Set, or the values in a Map
* E - An exception class
* T - A generic type

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#ibm-pcon)

**Generic types are not covariant**

A common source of confusion with generic types is to assume that, like arrays, they are covariant. They are not. This is a fancy way of saying that List<Object> is *not* a supertype of List<String>.

If A extends B, then an array of A is also an array of B, and you can freely supply an A[] where a B[] is expected:

|  |
| --- |
| Integer[] intArray = new Integer[10];  Number[] numberArray = intArray; |

The code above is valid because an Integer *is* a Number, and an Integer array *is* a Number array. However, the same is not true with generics. The following code is invalid:

|  |
| --- |
| List<Integer> intList = new ArrayList<Integer>();  List<Number> numberList = intList; // invalid |

At first, most Java programmers find this lack of covariance annoying, or even "broken," but there is a good reason for it. If you could assign a List<Integer> to a List<Number>, the following code would violate the type safety that generics are supposed to provide:

|  |
| --- |
| List<Integer> intList = new ArrayList<Integer>();  List<Number> numberList = intList; // invalid  numberList.add(new Float(3.1415)); |

Because intList and numberList are aliased, the above code, if allowed, would let you put things other than Integers intointList. However, there is a way to write flexible methods which can accept a family of generic types, as you'll see in the next panel.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#ibm-pcon)

**Type wildcards**

Suppose you have this method:

|  |
| --- |
| void printList(List l) {  for (Object o : l)  System.out.println(o);  } |

The code above compiles on JDK 5.0, but if you try to call it with a List<Integer>, you'll get a warning. The warning occurs because you're passing a generic type (List<Integer>) to a method that only promises to treat it as a List (a so-called *raw type*), which could undermine the type safety of using generics.

What if you try writing the method like this:

|  |
| --- |
| void printList(List<Object> l) {  for (Object o : l)  System.out.println(o);  } |

It still won't compile, because a List<Integer> is *not* a List<Object> (as you learned in the previous section, [Generic types are not covariant](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#notcovariant)). That's really annoying -- now your generic version is less useful than the original, nongeneric version!

The solution is to use a *type wildcard*:

|  |
| --- |
| void printList(List<?> l) {  for (Object o : l)  System.out.println(o);  } |

The question mark in the above code is a type wildcard. It is pronounced "unknown" (as in "list of unknown"). List<?> is a supertype of any generic List, so you can freely pass List<Object>, List<Integer>, or List<List<List<Flutzpah>>> toprintList().

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#ibm-pcon)

**Type wildcards in action**

The previous section, [Type wildcards](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#wcards), introduced the type wildcard, which lets you declare variables of type List<?>. What can you do with such a List? Quite conveniently, you can retrieve elements from it, but not add elements to it. The reason for this is not that the compiler knows which methods modify the list and which do not. It is that (most of) the mutative methods happen to require more type information than nonmutative methods. The following code works just fine:

|  |
| --- |
| List<Integer> li = new ArrayList<Integer>();  li.add(new Integer(42));  List<?> lu = li;  System.out.println(lu.get(0)); |

Why does this work? The compiler has no clue as to the value of the type parameter of List for lu. However, the compiler is smart enough that it can do some type inference. In this case, it infers that the unknown type parameter must extend Object . (This particular inference is no great leap, but the compiler can make some pretty impressive type inferences, as you will see later (in [The gory details](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html) ). So it lets you call List.get() and infers the return type to be Object.

On the other hand, the following code does not work:

|  |
| --- |
| List<Integer> li = new ArrayList<Integer>();  li.add(new Integer(42));  List<?> lu = li;  lu.add(new Integer(43)); // error |

In this case, the compiler cannot make a strong enough inference about the type parameter of List for lu to be certain that passing an Integer to List.add() is type-safe. So the compiler will not allow you to do this.

Lest you still think the compiler has some notion of which methods change the contents of the list and which don't, note that the following code will work, because it doesn't depend on the compiler having to know anything about the type parameter of lu:

|  |
| --- |
| List<Integer> li = new ArrayList<Integer>();  li.add(new Integer(42));  List<?> lu = li;  lu.clear(); |

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#ibm-pcon)

**Generic methods**

You have seen (in [Type parameters](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#typeparameters)) that a class can be made generic by adding a list of formal type arguments to its definition. Methods can also be made generic, whether or not the class in which they are defined is generic.

Generic classes enforce type constraints across multiple method signatures. In List<V>, the type parameter V appears in the signatures for get(), add(), contains(), etc. When you create a variable of type Map<K, V>, you are asserting a type constraint across methods. The values you pass to put() will be the same type as those returned by get().

Similarly, when you declare a generic method, you generally do so because you want to assert a type constraint across multiple arguments to the method. For example, depending on the boolean value of the first argument to the ifThenElse() method in the following code, it will return either the second or third argument:

|  |
| --- |
| public <T> T ifThenElse(boolean b, T first, T second) {  return b ? first : second;  } |

Note that you can call ifThenElse() without explicitly telling the compiler what value of T you want. The compiler doesn't need to be told explicitly what value T will have; it only knows that they must all be the same. The compiler allows you to call the following code, because the compiler can use type inference to infer that substituting String for T satisfies all type constraints:

|  |
| --- |
| String s = ifThenElse(b, "a", "b"); |

Similarly, you can call:

|  |
| --- |
| Integer i = ifThenElse(b, new Integer(1), new Integer(2)); |

However, the compiler doesn't allow the following code, because no type will satisfy the required type constraints:

|  |
| --- |
| String s = ifThenElse(b, "pi", new Float(3.14)); |

Why would you choose to use a generic method, instead of adding the type T to the class definition? There are (at least) two cases in which this makes sense:

* When the generic method is static, in which case class type parameters cannot be used.
* When the type constraints on T really are local to the method, which means that there is no constraint that the *same* typeT be used in *another* method signature of the same class. By making the type parameter for a generic method local to the method, you simplify the signature of the enclosing class.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#ibm-pcon)

**Bounded types**

In the example in the previous section, [Generic methods](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#gmethods), the type parameter V was an unconstrained, or *unbounded*, type. Sometimes you need to specify additional constraints on a type parameter, while still not specifying it completely.

Consider the example Matrix class, which uses a type parameter V that is bounded by the Number class:

|  |
| --- |
| public class Matrix<V extends Number> { ... } |

The compiler would allow you to create a variable of type Matrix<Integer> or Matrix<Float>, but would issue an error if you tried to define a variable of type Matrix<String>. The type parameter V is said to be *bounded by* Number. In the absence of a type bound, a type parameter is assumed to be bounded by Object . This is why the example in the previous section, [Generic methods](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#gmethods), allows List.get() to return an Object when called on a List<?>, even though the compiler doesn't know the type of

**A simple generic class**

**Writing a basic container class**

At this point you're ready to write a simple generic class. By far, the most common use cases for generic classes are container classes, such as the Collections framework, or value-holder classes, such as WeakReference or ThreadLocal. Let's write a class, similar to List, that acts as a container, using generics to express the constraint that all elements of the Lhist will have the same type. For simplicity of implementation, Lhist uses a fixed-size array to store values and does not accept null values.

The Lhist class will have one type parameter, V, which is the type of values in the Lhist, and will have the following methods:

|  |
| --- |
| public class Lhist<V> {  public Lhist(int capacity) { ... }  public int size() { ... }  public void add(V value) { ... }  public void remove(V value) { ... }  public V get(int index) { ... }  } |

To instantiate a Lhist, you simply specify the type parameter when declaring one, and the desired capacity:

|  |
| --- |
| Lhist<String> stringList = new Lhist<String>(10); |

**Implementing the constructor**

The first stumbling block you will run into when implementing the Lhist class is implementing the constructor. You'd like to implement it like this:

|  |
| --- |
| public class Lhist<V> {  private V[] array;  public Lhist(int capacity) {  array = new V[capacity]; // illegal  }  } |

This seems a natural way to allocate the backing array, but unfortunately you can't do this. The reasons why are complicated; you'll understand them later when you get to the topic of erasure in [The gory details](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html) . The way to do what you want is ugly and counterintuitive. One possible implementation of the constructor is this (which uses the approach taken by the Collections classes):

|  |
| --- |
| public class Lhist<V> {  private V[] array;  public Lhist(int capacity) {  array = (V[]) new Object[capacity];  }  } |

Alternatively, you could use reflection to instantiate the array. But doing so would require passing an additional argument to the constructor -- a *class literal*, such as Foo.class . Class literals will be discussed later as well, in the section on [Class<T>](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#classt) .

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section4.html#ibm-pcon)

**Implementing the methods**

Implementing the rest of Lhist's methods is a lot easier. Here's the full implementation of the Lhist class:

|  |
| --- |
| public class Lhist<V> {  private V[] array;  private int size;  public Lhist(int capacity) {  array = (V[]) new Object[capacity];  }  public void add(V value) {  if (size == array.length)  throw new IndexOutOfBoundsException(Integer.toString(size));  else if (value == null)  throw new NullPointerException();  array[size++] = value;  }  public void remove(V value) {  int removalCount = 0;  for (int i=0; i<size; i++) {  if (array[i].equals(value))  ++removalCount;  else if (removalCount > 0) {  array[i-removalCount] = array[i];  array[i] = null;  }  }  size -= removalCount;  }  public int size() { return size; }  public V get(int i) {  if (i >= size)  throw new IndexOutOfBoundsException(Integer.toString(i));  return array[i];  }  } |

Note that you use the formal type parameter V in methods that will accept or return a V, but you do not have any idea what methods or fields V has, because it is not known to the generic code.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section4.html#ibm-pcon)

**Using the Lhist class**

Using the Lhist class is easy. To define a Lhist of integers, you simply supply the actual value for the type parameter, in the declaration and the constructor:

|  |
| --- |
| Lhist<Integer> li = new Lhist<Integer>(30); |

The compiler knows that any value returned by li.get() will be of type Integer, and it will enforce that anything passed toli.add() or li.remove() will be an Integer. With the exception of the weird way the constructor was implemented, you didn't need to do anything terribly special to make Lhist a generic class.

**Generics in the Java class libraries**

**Collections classes**

By far, the biggest consumer of generics support in the Java class libraries is the Collections framework. Just as container classes were the primary motivation for templates in C++ (see the [Appendix A: Comparison to C++ templates](http://www.ibm.com/developerworks/java/tutorials/j-generics/section8.html#appendix)) (although they have been subsequently used for much more), improving the type safety of the Collection classes was the primary motivation for generics in the Java language. The Collections classes also serve as a model of how generics can be used, because they demonstrate almost all the standard tricks and idioms of generic types.

All of the standard collection interfaces are generified -- Collection<V>, List<V>, Set<V>, and Map<K,V>. Similarly, the implementations of the collection interfaces are generified with the same type arguments, so HashMap<K,V> implementsMap<K,V>, etc.

The Collections classes also use many of the "tricks" and idioms of generics, such as upper- and lower-bounded wildcards. For example, in the interface Collection<V>, the addAll method is defined as follows:

|  |
| --- |
| interface Collection<V> {  boolean addAll(Collection<? extends V>);  } |

This definition, which combines wildcard type parameters with bounded type parameters, allows you to add the contents of aCollection<Integer> to a Collection<Number>.

If the class libraries defined addAll() to take a Collection<V>, you would not be able to add the contents of aCollection<Integer> to a Collection<Number>. Rather than restricting the parameter of addAll() to be a collection containing exactly the same type as the collection you are adding to, it is possible to instead make the more reasonable constraint that the *elements* of the collection being passed to addAll() be suitable for addition to your collection. Bounded types let you do that, and the use of bounded wildcards frees you from the requirement to make up another placeholder name that will not be used anywhere else.

As a subtle example of how generifying a class can change its semantics (if you're not careful), notice that the type of the argument to Collection.removeAll() is Collection<?>, not Collection<? extends V>. This is because it is acceptable to pass a collection of mixed type to removeAll(), and defining removeAll more restrictively would have altered the semantics and usefulness of the method. This illustrates how generifying an existing class is a lot harder than defining a new generic class, because you must be careful not to change the semantics of the class or break existing nongeneric code.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Other container classes**

In addition to the Collections classes, several other classes in the Java class library act as containers for values. These classes include WeakReference, SoftReference, and ThreadLocal. They have all been generified over the type of value they are a container for, so WeakReference<T> is a weak reference to an object of type T, and ThreadLocal<T> is a handle to a thread-local variable of type T.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Generics are not just for containers**

The most common, and most straightforward, use for generic types is container classes, such as the Collections classes or the references classes (such as WeakReference<T>.) The meaning of the type parameter in Collection<V> is intuitively obvious -- "a collection of values all of which are of type V." Similarly, ThreadLocal<T> has an obvious interpretation -- "a thread-local variable whose type is T." However, nothing in the specification of generics has anything to do with containment.

The meaning of the type parameter in classes such as Comparable<T> or Class<T> is more subtle. Sometimes, as in the case of Class<T>, the type variable is there mostly to help the compiler with type inference. Sometimes, as in the case of the crypticEnum<E extends Enum<E>>, it is there to place a constraint on the class hierarchy's structure.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Comparable<T>**

The Comparable interface has been generified so that an object that implements Comparable declares what type it can be compared with. (Usually, this is the type of the object itself, but sometimes might be a superclass.)

|  |
| --- |
| public interface Comparable<T> {  public boolean compareTo(T other);  } |

So the Comparable interface includes a type parameter T, which is the type of object a class implementing Comparable can be compared to. This means that if you are defining a class that implements Comparable, such as String, you must declare not only that the class supports comparison, but also what it is comparable to, which is usually itself:

|  |
| --- |
| public class String implements Comparable<String> { ... } |

Now consider an implementation of a binary max() method. You want to take two arguments of the same type, both to beComparable, and to be Comparable to each other. Fortunately, that is relatively straightforward if you use a generic method and a bounded type parameter:

|  |
| --- |
| public static <T extends Comparable<T>> T max(T t1, T t2) {  if (t1.compareTo(t2) > 0)  return t1;  else  return t2;  } |

In this case, you define a generic method, generified over a type T, that you constrain to extend (implement) Comparable<T>. Both of the arguments must be of type T, which means they are the same type, support comparison, and are comparable to each other. Easy!

Even better, the compiler will use type inference to decide what value of T was meant when calling max(). So the following invocation works without having to specify T at all:

|  |
| --- |
| String s = max("moo", "bark"); |

The compiler will figure out that the intended value of T is String, and it will compile and type-check accordingly. But if you tried to call max() with arguments of class X that didn't implement Comparable<X>, the compiler wouldn't permit it.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Class<T>**

The class Class has been generified, but in a way that many people find confusing at first. What is the meaning of the type parameter T in Class<T>? It turns out that it is the class instance being referenced. How can that be? Isn't that circular? And even if not, why would it be defined that way?

In prior JDKs, the definition of the Class.newInstance() method returned Object, which you would then likely cast to another type:

|  |
| --- |
| class Class {  Object newInstance();  } |

However, using generics, you define the Class.newInstance() method with a more specific return type:

|  |
| --- |
| class Class<T> {  T newInstance();  } |

How do you create an instance of type Class<T>? Just as with nongeneric code, you have two ways: calling the methodClass.forName() or using the class literal X.class. Class.forName() is defined to return Class<?>. On the other hand, the class literal X.class is defined to have type Class<X>, so String.class is of type Class<String>.

What is the benefit of having Foo.class be of type Class<Foo>? The big benefit is that it can improve the type safety of code that uses reflection, through the magic of type inference. Also, you don't need to cast Foo.class.newInstance() to Foo.

Consider a method that retrieves a set of objects from a database and returns a collection of JavaBeans objects. You instantiate and initialize the created objects via reflection, but that doesn't mean type safety has to go totally out the window. Consider this method:

|  |
| --- |
| public static<T> List<T> getRecords(Class<T> c, Selector s) {  // Use Selector to select rows  List<T> list = new ArrayList<T>();  for (/\* iterate over results \*/) {  T row = c.newInstance();  // use reflection to set fields from result  list.add(row);  }  return list;  } |

You can call this method simply like this:

|  |
| --- |
| List<FooRecord> l = getRecords(FooRecord.class, fooSelector); |

The compiler will infer the return type of getRecords() from the fact that FooRecord.class is of type Class<FooRecord>. You use the class literal both to construct the new instance and to provide type information to the compiler for it to use in type checking.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Replacing T[] with Class<T>**

The Collection interface includes a method for copying the contents of a collection into an array of a caller-specified type:

|  |
| --- |
| public Object[] toArray(Object[] prototypeArray) { ... } |

The semantics of toArray(Object[]) are that if the passed array is big enough, it should be used to store the results; otherwise a new array of the same type is allocated, using reflection. In general, passing an array as a parameter solely to provide the desired return type is kind of a cheap trick, but prior to the addition of generics, it was the most convenient way to communicate type information to a method.

With generics, you have a more straightforward way to do this. Instead of defining toArray() as above, a generic toArray()might look like this:

|  |
| --- |
| public<T> T[] toArray(Class<T> returnType) |

Invoking such a toArray() method is simple:

|  |
| --- |
| FooBar[] fba = something.toArray(FooBar.class); |

The Collection interface has not been changed to use this technique, because that would break many existing collection implementations. But if Collection were being rebuilt with generics from the ground up, it would almost certainly use this idiom for specifying which type it wants the return value to be.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Enum<E>**

One of the other additions to the Java language in JDK 5.0 is enumerations. When you declare an enumeration with the enumkeyword, the compiler internally generates a class for you that extends Enum and declares static instances for each value of the enumeration. So if you say:

|  |
| --- |
| public enum Suit {HEART, DIAMOND, CLUB, SPADE}; |

the compiler will internally generate a class called Suit, which extends java.lang.Enum<Suit> and has constant (public static final) members called HEART, DIAMOND, CLUB, and SPADE, each of which is of the Suit class.

Like Class, Enum is a generic class. But unlike Class, its signature is a little more complicated:

|  |
| --- |
| class Enum<E extends Enum<E>> { . . . } |

What on earth does that mean? Doesn't that lead to an infinite recursion?

Let's take it in steps. The type parameter E is used in various methods of Enum, such as compareTo() orgetDeclaringClass(). In order for these to be type-safe, the Enum class must be generified over the class of the enumeration.

So what about the extends Enum<E> part? That has two parts too. The first part says that classes that are type arguments toEnum must themselves be subtypes of Enum, so you can't declare a class X to extend Enum<Integer>. The second part says that any class that extends Enum must pass *itself* as the type parameter. You cannot declare X to extend Enum<Y>, even if Y extendsEnum.

In summary, Enum is a parameterized type that can only be instantiated for its subtypes, and those subtypes will then inherit methods that depend on the subtype. Whew! Fortunately, in the case of Enum, the compiler does the work for you, and the right thing happens.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Interoperating with nongeneric code**

Millions of lines of existing code use classes from the Java class library that have been generified, such as the Collections framework, Class, and ThreadLocal. It is important that the improvements in JDK 5.0 not break all that code, so the compiler allows you to use generic classes without specifying their type parameters.

Of course, doing things "the old way" is less safe than the new way, because you are bypassing the type safety that the compiler is ready to offer you. If you try to pass a List<String> to a method that accepts a List, it will work, but the compiler will emit a warning that type safety might be lost (a so-called "unchecked conversion" warning.)

A generic type without type parameters, such as a variable declared to be of type List instead of List<Something>, is referred to as a *raw type*. A raw type is assignment compatible with any instantiation of the parameterized type, but such an assignment will generate an unchecked-conversion warning.

To eliminate *some* of the unchecked-conversion warnings, assuming you are not ready to generify all your code, you can use a wildcard type parameter instead. Use List<?> instead of List. List is a raw type; List<?> is a generic type with an unknown type parameter. The compiler will treat them differently and likely emit fewer warnings.

In any case, the compiler will generate casts when it generates bytecode, so in no case will the generated bytecode be any less safe than it would be without generics. If you manage to subvert type safety by using raw types or playing games with class files, you will get the same ClassCastExceptions or ArrayStoreExceptions you would have gotten without generics.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#ibm-pcon)

**Checked collections**

As an aid to migrating from raw collection types to generic collection types, the Collections framework adds some new collection wrappers to provide early warnings for some type-safety bugs. Just as the Collections.unmodifiableSet() factory method wraps an existing Set with a Set that does not permit any modification, the Collections.checkedSet() (alsocheckedList() and checkedMap()) factory methods create a wrapper, or view, class that prevents you from placing variables of the wrong type in a collection.

The checkedXxx() methods all take a class literal as an argument, so they can check (at runtime) that modifications are allowable. A typical implementation would look like this:

|  |
| --- |
| public class Collections {  public static <E> Collection<E>  checkedCollection(Collection<E> c, Class<E> type ) {  return new CheckedCollection<E>(c, type);  }  private static class CheckedCollection<E> implements Collection<E> {  private final Collection<E> c;  private final Class<E> type;  CheckedCollection(Collection<E> c, Class<E> type) {  this.c = c;  this.type = type;  }  public boolean add(E o) {  if (!type.isInstance(o))  throw new ClassCastException();  else  return c.add(o);  }  }  } |

**The gory details**

**Erasure**

Perhaps the most challenging aspect of generic types is *erasure*, which is the technique underlying the implementation of generics in the Java language. Erasure means that the compiler basically throws away much of the type information of a parameterized class when generating the class file. The compiler generates code with casts in it, just as programmers did by hand before generics. The difference is that the compiler has first validated a number of type-safety constraints that it could not have validated without generic types.

The implications of implementing generics through erasure are considerable and, at first, confusing. Although you cannot assign a List<Integer> to a List<Number> because they are different types, variables of type List<Integer> andList<Number> are of the same class! To see this, try evaluating this expression:

|  |
| --- |
| new List<Number>().getClass() == new List<Integer>().getClass() |

The compiler generates only one class for List. By the time the bytecode for List is generated, little trace of its type parameter remains.

When generating bytecode for a generic class, the compiler replaces type parameters with their *erasure*. For an unbounded type parameter (<V>), its erasure is Object. For an upper-bounded type parameter (<K extends Comparable<K>>), its erasure is the erasure of its upper bound (in this case, Comparable). For type parameters with multiple bounds, the erasure of its leftmost bound is used.

If you inspected the generated bytecode, you would not be able to tell the difference between code that came fromList<Integer> and List<String>. The type bound T is replaced in the bytecode with T's upper bound, which is usuallyObject.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Implications of erasure**

Erasure has a number of implications that might seem odd at first. For example, because a class can implement an interface only once, you cannot define a class like this:

|  |
| --- |
| // invalid definition  class DecimalString implements Comparable<String>, Comparable<Integer> { ... } |

In light of erasure, the above declaration simply does not make sense. The two instantiations of Comparable are the same interface, and they specify the same compareTo() method. You cannot implement a method or an interface twice.

Another, much more annoying implication of erasure is that you cannot instantiate an object or an array using a type parameter. This means you can't use new T() or new T[10] in a generic class with a type parameter T. The compiler simply does not know what bytecode to generate.

There are some workarounds for this issue, generally involving reflection and the use of class literals (Foo.class), but they are annoying. The constructor in the Lhist example class displayed one such technique for working around the problem (see[Implementing the constructor](http://www.ibm.com/developerworks/java/tutorials/j-generics/section4.html#constructor)), and the discussion of toArray() (in [Replacing T[] with Class<T>](http://www.ibm.com/developerworks/java/tutorials/j-generics/section5.html#replacing)) offered another.

Another implication of erasure is that it makes no sense to use instanceof to test if a reference is an instance of a parameterized type. The runtime simply cannot tell a List<String> from a List<Number>, so testing for (x instanceof List<String>) doesn't make any sense.

Similarly, the following method won't increase the type safety of your programs:

|  |
| --- |
| public <T> T naiveCast(T t, Object o) { return (T) o; } |

The compiler will simply emit an unchecked warning, because it has no idea whether the cast is safe or not.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Types versus classes**

The addition of generic types has made the type system in the Java language more complicated. Previously, the language had two kinds of types -- reference types and primitive types. For reference types, the concepts of *type* and *class* were basically interchangeable, as were the terms *subtype* and *subclass*.

With the addition of generics, the relationship between type and class has become more complex. List<Integer> andList<Object> are distinct types, but they are of the same class. Even though Integer extends Object, a List<Integer> is not a List<Object>, and it cannot be assigned or even cast to List<Object>.

On the other hand, now there is a new weird type called List<?>, which is a supertype of both List<Integer> andList<Object>. And there is the even weirder List<? extends Number>. The structure and shape of the type hierarchy got a lot more complicated. Types and classes are no longer mostly the same thing.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Covariance**

As you learned earlier (see [Generic types are not covariant](http://www.ibm.com/developerworks/java/tutorials/j-generics/section3.html#notcovariant)), generic types, unlike arrays, are not covariant. An Integer is aNumber, and an array of Integer is an array of Number. Therefore, you can freely assign an Integer[] reference to a variable of type Number[]. But a List<Integer> is not a List<Number>, and for good reason -- the ability to assign a List<Integer> to a List<Number> could subvert the type checking that generics are supposed to provide.

This means that if you have a method argument that is a generic type, such as Collection<V>, you cannot pass a collection of a subclass of V to that method. If you want to give yourself the freedom to do so, you must use bounded type parameters, such as Collection<T extends V> (or Collection<? extends V>.)

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Arrays**

You can use generic types in most situations where you could use a nongeneric type, but there are some restrictions. For example, you cannot declare an array of a generic type (except if the type arguments are unbounded wildcards). The following code is illegal:

|  |
| --- |
| List<String>[] listArray = new List<String>[10]; // illegal |

Permitting such a construction could create problems, because arrays in Java language are covariant, but parameterized types are not. Because any array type is type-compatible with Object[] (a Foo[] *is* an Object[]), the following code would compile without warning, but it would fail at runtime, which would undermine the goal of having any program that compiles without unchecked warnings be type-safe:

|  |
| --- |
| List<String>[] listArray = new List<String>[10]; // illegal  Object[] oa = listArray;  oa[0] = new List<Integer>();  String s = lsa[0].get(0); // ClassCastException |

If, on the other hand, listArray were of type List<?>, an explicit cast would be required in the last line. Although it would still generate a runtime error, it would not undermine the type-safety guarantees offered by generics (because the error would be in the explicit cast). So arrays of List<?> are permitted.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**New meanings for extends**

Before the introduction of generics in the Java language, the extends keyword always meant that a new class or interface was being created that inherited from another class or interface.

With the introduction of generics, the extends keyword has another meaning. You use extends in the definition of a type parameter (Collection<T extends Number>) or a wildcard type parameter (Collection<? extends Number>).

When you use extends to denote a type parameter bound, you are not requiring a subclass-superclass relationship, but merely a subtype-supertype relationship. It is also important to remember that the bounded type does not need to be a strict subtype of the bound; it could *be* the bound as well. In other words, for a Collection<? extends Number>, you could assign aCollection<Number> (although Number is not a strict subtype of Number) as well as a Collection<Integer>,Collection<Long>, Collection<Float>, and so on.

In any of these meanings, the type on the right-hand side of extends can be a parameterized type (Set<V> extends Collection<V>).

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Bounded types**

So far, you've seen one kind of type bound -- the *upper bound*. Specifying an upper bound constrains a type parameter to be a supertype of (or equal to) a given type bound, as in Collection<? extends Number>. It is also possible, though less common, to specify a *lower bound*, which you write as Collection<? super Foo>. Only wildcards can have lower bounds.

In addition to specifying a type constraint on the type parameter, specifying a bound has another significant effect. If a type T is known to extend Number, then the methods and fields of Number can be accessed through a variable of type T. It might not be known at compile time what the value of T is, but it is known at least to be a Number.

There are some restrictions on which classes can act as type bounds. Primitive types and array types cannot be used as type bounds (but array types can be used as wildcard bounds). Any reference type (including parameterized types) can be used as a type bound.

|  |
| --- |
| class C <T extends int> // illegal  class C <T extends Foo[]> // illegal  class C <T extends Foo> //legal  class C <T extends Foo<? extends Moo<T>>> //legal  class C <T, V extends T> // legal |

One place where you might use a lower bound is in a method that selects elements from one collection and puts them in another. For example:

|  |
| --- |
| class Bunch<V> {  public void add(V value) { ... }  public void copyTo(Collection<? super V>) { ... }  ...  } |

The copyTo() method copies all the values from the Bunch into a specified collection. Rather than specify that it must be aCollection<V>, you can specify that it be a Collection<? super V>, which means copyTo() can copy the contents of aBunch<String> to a Collection<Object> or a Collection<String>, rather than just a Collection<String>.

The other common case for lower bounds is with the Comparable interface. Rather than specifying:

|  |
| --- |
| public static <T extends Comparable<T>> T max(Collection<T> c) { ... } |

You can be more flexible in what types you accept:

|  |
| --- |
| public static <T extends Comparable<?  super T>> T max(Collection<T> c) { ... } |

This way, you can pass a type that is comparable to its supertype, in addition to a type that is comparable to itself, for some additional flexibility. This becomes valuable for classes that extend classes that are already Comparable:

|  |
| --- |
| public class Base implements Comparable<Base> { ... }  public class Child extends Base { } |

Because Child already implements Comparable<Base> (which it inherits from the superclass Base), you can pass it to the second example of max() above, but not the first.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Multiple bounds**

A type parameter can have more than one bound. This is useful when you want to constrain a type parameter to be, say, bothComparable and Serializable. The syntax for multiple bounds is to separate the bounds with an ampersand:

|  |
| --- |
| class C<T extends Comparable<? super T> & Serializable> |

A wildcard type can have a single bound -- either an upper or a lower bound. A named type parameter can have one or more upper bounds. A type parameter with multiple bounds can be used to access the methods and fields of each of its bounds.

[**Back to top**](http://www.ibm.com/developerworks/java/tutorials/j-generics/section6.html#ibm-pcon)

**Type parameters and type arguments**

In the definition of a parameterized class, the placeholder names (such as V in Collection<V>) are referred to as *type parameters*. They have a similar role to that of formal arguments in a method definition. In a declaration of a variable of a parameterized class, the type values specified in the declaration are referred to as *type arguments*. These have a role similar to actual arguments in a method call. So given the definition:

|  |
| --- |
| interface Collection<V> { ... } |

and the declaration:

|  |
| --- |
| Collection<String> cs = new HashSet<String>(); |

the name V (which can be used throughout the body of the Collection interface) is called a type parameter. In the declaration ofcs, both usages of String are type arguments (one for Collection<V> and the other for HashSet<V>.)

There are some restrictions on when you can use type parameters. Most of the time, you can use them anyplace you can use an actual type definition. But there are exceptions. You cannot use them to create objects or arrays, and you cannot use them in a static context or in the context of handling an exception. You also cannot use them as supertypes (class Foo<T> extends T), in instanceof expressions, or as class literals.

Similarly, there are some restrictions on which types you can use as type arguments. They must be reference types (not primitive types), wildcards, type parameters, or instantiations of other parameterized types. So you can define a List<String> (reference type), a List<?> (wildcard), or a List<List<?>> (instantiation of other parameterized types). Inside the definition of a parameterized type with type parameter T, you could also declare a List<T> (type parameter.)

**Wrap-up**

**Summary**

The addition of generic types is a major change to both the Java language and the Java class libraries. Generic types (generics) can improve the type safety, maintainability, and reliability of Java applications, but at the cost of some additional complexity.

Great care was taken to ensure that existing classes will continue to work with the generified class libraries in JDK 5.0, so you can get started with generics as quickly or as slowly as you like.