O'Reilly Book Excerpts: Java in a Nutshell, 5th Edition

Editor's note: In part one of this two-part excerpt from Java in a Nutshell, 5th Edition, *David Flanagan described how to use generic types. This week David details how to write your own generic types and generic methods, and concludes with a tour of important generic types in the core Java API.*

Writing Generic Types and Methods

Creating a simple generic type is straightforward. First, declare your type variables by enclosing a comma-separated list of their names within angle brackets after the name of the class or interface. You can use those type variables anywhere a type is required in any instance fields or methods of the class. Remember, though, that type variables exist only at compile time, so you can't use a type variable with the runtime operators `instanceof` and `new`.

We begin this section with a simple generic type, which we will subsequently refine. This code defines a `Tree` data structure that uses the type variable `V` to represent the type of the value held in each node of the tree:

```
import java.util.*;

/**
 * A tree is a data structure that holds values of type V.
 * Each tree has a single value of type V and can have any number of
 * branches, each of which is itself a Tree.
 */
public class Tree<V> {
    // The value of the tree is of type V.
    V value;

    // A Tree<V> can have branches, each of which is also a Tree<V>

    List<Tree<V>> branches = new ArrayList<Tree<V>>();

    // Here's the constructor.  Note the use of the type variable V.
    public Tree(V value) { this.value = value; }

    // These are instance methods for manipulating the node value and branches.
    // Note the use of the type variable V in the arguments or return types.
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<V> branch) { branches.add(branch); }
}
```

As you've probably noticed, the naming convention for type variables is to use a single capital letter. The use of a single letter distinguishes these variables from the names of actual types since real-world types always have longer, more descriptive names. The use of a capital letter is consistent with type naming conventions and distinguishes type variables from local variables, method parameters, and fields, which are sometimes written with a single lowercase letter. Collection classes like those in `java.util` often use the type variable `E` for "Element type." When a type variable can represent absolutely anything, `T` (for Type) and `S` are used as the most generic type variable names possible (like using `i` and `j` as loop variables).

Notice that the type variables declared by a generic type can be used only by the instance fields and methods (and nested types) of the type and not by static fields and methods. The reason, of course, is that it is instances of generic types that are parameterized. Static members are shared by all instances and parameterizations of the class, so static members do not have type parameters associated with them. Methods,

including static methods, can declare and use their own type parameters, however, and each invocation of such a method can be parameterized differently. We'll cover this later in the chapter.

Type variable bounds

The type variable `V` in the declaration above of the `Tree<V>` class is unconstrained: `Tree` can be parameterized with absolutely any type. Often we want to place some constraints on the type that can be used: we might want to enforce that a type parameter implements one or more interfaces, or that it is a subclass of a specified class. This can be done by specifying a *bound* for the type variable. We've already seen upper bounds for wildcards, and upper bounds can also be specified for type variables using a similar syntax. The following code is the `Tree` example rewritten to make `Tree` objects `Serializable` and `Comparable`. In order to do this, the example uses a type variable bound to ensure that its value type is also `Serializable` and `Comparable`. Note how the addition of the `Comparable` bound on `V` enables us to write the `compareTo()` method `Tree` by guaranteeing the existence of a `compareTo()` method on `V`. [4]

```
import java.io.Serializable;
import java.util.*;

public class Tree<V extends Serializable & Comparable<V>>
    implements Serializable, Comparable<Tree<V>>

{
    V value;
    List<Tree<V>> branches = new ArrayList<Tree<V>>();

    public Tree(V value) { this.value = value; }

    // Instance methods
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<V> branch) { branches.add(branch); }

    // This method is a nonrecursive implementation of Comparable<Tree<V>>

    // It only compares the value of this node and ignores branches.
    public int compareTo(Tree<V> that) {
        if (this.value == null && that.value == null) return 0;
        if (this.value == null) return -1;
        if (that.value == null) return 1;
        return this.value.compareTo(that.value);
    }

    // javac -Xlint warns us if we omit this field in a Serializable class
    private static final long serialVersionUID = 833546143621133467L;
}
```

The bounds of a type variable are expressed by following the name of the variable with the word `extends` and a list of types (which may themselves be parameterized, as `Comparable` is). Note that with more than one bound, as in this case, the bound types are separated with an ampersand rather than a comma. Commas are used to separate type variables and would be ambiguous if used to separate type variable bounds as well. A type variable can have any number of bounds, including any number of interfaces and at most one class.

Wildcards in generic types

Earlier in the chapter we saw examples using wildcards and bounded wildcards in methods that manipulated parameterized types. They are also useful in generic types. Our current design of the `Tree` class requires the

value object of every node to have exactly the same type, `V`. Perhaps this is too strict, and we should allow branches of a tree to have values that are a subtype of `V` instead of requiring `V` itself. This version of the `Tree` class (minus the `Comparable` and `Serializable` implementation) is more flexible:

```
public class Tree<V> {
    // These fields hold the value and the branches
    V value;
    List<Tree<? extends V>> branches = new ArrayList<Tree<? extends V>>();

    // Here's a constructor
    public Tree(V value) { this.value = value; }

    // These are instance methods for manipulating value and branches
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<? extends V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<? extends V> branch) { branches.add(branch); }
}
```

The use of bounded wildcards for the branch type allow us to add a `Tree<Integer>`, for example, as a branch of a `Tree<Number>`:

```
Tree<Number> t = new Tree<Number>(0);  // Note autoboxing
t.addBranch(new Tree<Integer>(1));     // int 1 autoboxed to Integer
```

If we query the branch with the `getBranch( )` method, the value type of the returned branch is unknown, and we must use a wildcard to express this. The next two lines are legal, but the third is not:

```
Tree<? extends Number> b = t.getBranch(0);
Tree<?> b2 = t.getBranch(0);
Tree<Number> b3 = t.getBranch(0);  // compilation error
```

When we query a branch like this, we don't know the precise type of the value, but we do still have an upper bound on the value type, so we can do this:

```
Tree<? extends Number> b = t.getBranch(0);
Number value = b.getValue();
```

What we cannot do, however, is set the value of the branch, or add a new branch to that branch. As explained earlier in the chapter, the existence of the upper bound does not change the fact that the value type is unknown. The compiler does not have enough information to allow us to safely pass a value to `setValue()` or a new branch (which includes a value type) to `addBranch()`. Both of these lines of code are illegal:

```
b.setValue(3.0); // Illegal, value type is unknown
b.addBranch(new Tree<Double>(Math.PI));
```

This example has illustrated a typical trade-off in the design of a generic type: using a bounded wildcard made the data structure more flexible but reduced our ability to safely use some of its methods. Whether or not this was a good design is probably a matter of context. In general, generic types are more difficult to design well. Fortunately, most of us will use the preexisting generic types in the `java.util` package much more frequently than we will have to create our own.

Generic methods

As noted earlier, the type variables of a generic type can be used only in the instance members of the type, not in the static members. Like instance methods, however, static methods can use wildcards. And although

static methods cannot use the type variables of their containing class, they can declare their own type variables. When a method declares its own type variable, it is called a *generic method*.

Here is a static method that could be added to the `Tree` class. It is not a generic method but uses a bounded wildcard much like the `sumList()` method we saw earlier in the chapter:

```
/** Recursively compute the sum of the values of all nodes on the tree */
public static double sum(Tree<? extends Number> t) {
    double total = t.value.doubleValue();
    for(Tree<? extends Number> b : t.branches) total += sum(b);
    return total;
}
```

This method could also be rewritten as a generic method by declaring a type variable to express the upper bound imposed by the wildcard:

```
public static <N extends Number> double sum(Tree<N> t) {
    N value = t.value;
    double total = value.doubleValue();
    for(Tree<? extends N> b : t.branches) total += sum(b);
    return total;
}
```

The generic version of `sum()` is no simpler than the wildcard version and the declaration of the type variable does not gain us anything. In a case like this, the wildcard solution is typically preferred over the generic solution. Generic methods are required where a single type variable is used to express a relationship between two parameters or between a parameter and a return value. The following method is an example:

```
// This method returns the largest of two trees, where tree size
// is computed by the sum() method.  The type variable ensures that
// both trees have the same value type and that both can be passed to sum().
public static <N extends Number> Tree<N> max(Tree<N> t, Tree<N> u) {
    double ts = sum(t);
    double us = sum(u);
    if (ts > us) return t;
    else return u;
}
```

This method uses the type variable `N` to express the constraint that both arguments and the return value have the same type parameter and that that type parameter is `Number` or a subclass.

It could be argued that constraining both arguments to have the same value type is too restrictive and that we should be allowed to call the `max( )` method on a `Tree<Integer>` and a `Tree<Double>`. One way to express this is to use two unrelated type variables to represent the two unrelated value types. Note, however, that we cannot use either variable in the return type of the method and must use a wildcard there:

```
public static <N extends Number, M extends Number>
    Tree<? extends Number> max(Tree<N> t, Tree<M> u) {...}
```

Since the two type variables `N` and `M` have no relation to each other, and since each is used in only a single place in the signature, they offer no advantage over bounded wildcards. The method is better written this way:

```
public static Tree<? extends Number> max(Tree<? extends Number> t,
                                         Tree<? extends Number> u) {...}
```

All the examples of generic methods shown here have been `static` methods. This is not a requirement: instance methods can declare their own type variables as well.

Invoking generic methods

When you use a generic type, you must specify the actual type parameters to be substituted for its type variables. The same is not generally true for generic methods: the compiler can almost always figure out the correct parameterization of a generic method based on the arguments you pass to the method. Consider the `max()` method defined above, for instance:

```
public static <N extends Number> Tree<N> max(Tree<N> t, Tree<N> u) {...}
```

You need not specify `N` when you invoke this method because `N` is implicitly specified in the values of the method arguments *t* and *u*. In the following code, for example, the compiler determines that `N` is `Integer`:

```
Tree<Integer> x = new Tree<Integer>(1);
Tree<Integer> y = new Tree<Integer>(2);
Tree<Integer> z = Tree.max(x, y);
```

The process the compiler uses to determine the type parameters for a generic method is called *type inference*. Type inference is relatively intuitive to understand, but the actual algorithm the compiler must use is surprisingly complex and is well beyond the scope of this book. Complete details are in Chapter 15 of *The Java Language Specification, Third Edition*.

Let's look at a slightly more complex version of type inference. Consider this method:

```
public class Util {
    /** Set all elements of a to the value v; return a. */
    public static <T> T[] fill(T[] a, T v) {
        for(int i = 0; i < a.length; i++) a[i] = v;
        return a;
    }
}
```

Here are two invocations of the method:

```
Boolean[] booleans = Util.fill(new Boolean[100], Boolean.TRUE);
Object o = Util.fill(new Number[5], new Integer(42));
```

In the first invocation, the compiler can easily determine that `T` is `Boolean`. In the second invocation, the compiler determines that `T` is `Number`.

In very rare circumstances you may need to explicitly specify the type parameters for a generic method. This is sometimes necessary, for example, when a generic method expects no arguments. Consider the `java.util.Collections.emptySet( )` method: it returns a set with no elements, but unlike the `Collections.singleton( )` method (you can look these up in the reference section), it takes no arguments that would specify the type parameter for the returned set. You can specify the type parameter explicitly by placing it in angle brackets *before* the method name:

```
Set<String> empty = Collections.<String>emptySet();
```

Type parameters cannot be used with an unqualified method name: they must follow a dot or come after the keyword `new` or before the keyword `this` or `super` used in a constructor.

It turns out that if you assign the return value of `Collections.emptySet()` to a variable, as we did above the type inference mechanism is able to infer the type parameter based on the variable type. Although the explicit type parameter specification in the code above can be a helpful clarification, it is not necessary and the line could be rewritten as:

```
Set<String> empty = Collections.emptySet();
```

An explicit type parameter is necessary when you use the return value of the `emptySet( )` method within a method invocation expression. For example, suppose you want to call a method named `printWords( )` that expects a single argument of type `Set<String>`. If you want to pass an empty set to this method, you could use this code:

```
printWords(Collections.<String>emptySet());
```

In this case, the explicit specification of the type parameter `String` is required.

Generic methods and arrays

Earlier in the chapter we saw that the compiler does not allow you to create an array whose type is parameterized. This is not, however, a restriction on all uses of arrays with generics. Consider the `Util.fill()` method defined above, for example. Its first argument and its return value are both of type `T[]`. The body of the method does not have to create an array whose element type is `T`, so the method is perfectly legal.

If you write a method that uses varargs (see Section 2.6.4 in Chapter 2) and a type variable, remember that invoking a varargs method performs an implicit array creation. Consider this method:

```
/** Return the largest of the specified values or null if there are none */
public static <T extends Comparable<T>> T max(T... values) { ...  }
```

You can invoke this method with parameters of type `Integer` because the compiler can insert the necessary array creation code for you when you call it. But you cannot call the method if you've cast the same arguments to be type `Comparable<Integer>` because it is not legal to create an array of type `Comparable<Integer>[ ]`.

Parameterized exceptions

Exceptions are thrown and caught at runtime, and there is no way for the compiler to perform type checking to ensure that an exception of unknown origin matches type parameters specified in a `catch` clause. For this reason, `catch` clauses may not include type variables or wildcards. Since it is not possible to catch an exception at runtime with compile-time type parameters intact, you are not allowed to make any subclass of `Throwable` generic. Parameterized exceptions are simply not allowed.

You can, however, use a type variable in the `throws` clause of a method signature. Consider this code, for example:

```
public interface Command<X extends Exception> {
    public void doit(String arg) throws X;
}
```

This interface represents a "command": a block of code with a single string argument and no return value. The code may throw an exception represented by the type parameter `X`. Here is an example that uses a parameterization of this interface:

```
Command<IOException> save = new Command<IOException>() {
    public void doit(String filename) throws IOException {
        PrintWriter out = new PrintWriter(new FileWriter(filename));
        out.println("hello world");
        out.close();
    }
```

```
};

try { save.doit("/tmp/foo");  }
catch(IOException e) { System.out.println(e); }
```

Generics Case Study: Comparable and Enum

The new generics features in Java 5.0 are used in the Java 5.0 APIs, most notably in `java.util` but also in `java.lang`, `java.lang.reflect`, and `java.util.concurrent`. These APIs were carefully created or reviewed by the inventors of generic types, and we can learn a lot about the good design of generic types and methods through the study of these APIs.

The generic types of `java.util` are relatively easy: for the most part they are collections classes, and type variables are used to represent the element type of the collection. Several important generic types in `java.lang` are more difficult. They are not collections, and it is not immediately apparent why they have been made generic. Studying these difficult generic types gives us a deeper understanding of how generics work and introduces some concepts that we have not yet covered in this chapter. Specifically, we'll examine the `Comparable` interface and the `Enum` class (the supertype of enumerated types, described later in this chapter) and will learn about an important but infrequently used feature of generics known as lower-bounded wildcards.

In Java 5.0, the `Comparable` interface has been made generic, with a type variable that specifies what a class is comparable to. Most classes that implement `Comparable` implement it on themselves. Consider `Integer`:

```
public final class Integer extends Number implements Comparable<Integer>
```

The raw `Comparable` interface is problematic from a type-safety standpoint. It is possible to have two `Comparable` objects that cannot be meaningfully compared to each other. Prior to Java 5.0, the nongeneric `Comparable` interface was useful but not fully satisfactory. The generic version of this interface, however, captures exactly the information we want: it tells us that a type is comparable and tells us what we can compare it to.

Now consider subclasses of comparable classes. `Integer` is `final` and cannot be subclassed, so let's look at `java.math.BigInteger` instead:

```
public class BigInteger extends Number implements Comparable<BigInteger>
```

If we implement a `BiggerInteger` subclass of `BigInteger`, it inherits the `Comparable` interface from its superclass. But note that it inherits `Comparable<BigInteger>` and not `Comparable<BiggerInteger>`. This means that `BigInteger` and `BiggerInteger` objects are mutually comparable, which is usually a good thing. `BiggerInteger` can override the `compareTo( )` method of its superclass, but it is not allowed to implement a different parameterization of `Comparable`. That is, `BiggerInteger` cannot both extend `BigInteger` and implement `Comparable<BiggerInteger>`. (In general, a class is not allowed to implement two different parameterizations of the same interface: we cannot define a type that implements both `Comparable<Integer>` and `Comparable<String>`, for example.)

When you're working with comparable objects (as you do when writing sorting algorithms, for example), remember two things. First, it is not sufficient to use `Comparable` as a raw type: for type safety, you must also specify what it is comparable to. Second, types are not always comparable to themselves: sometimes they're comparable to one of their ancestors. To make this concrete, consider the `java.util.Collections.max()` method:

```
public static <T extends Comparable<? super T>> T max(Collection<? extends T> c)
```

This is a long, complex generic method signature. Let's walk through it:

- The method has a type variable `T` with complicated bounds that we'll return to later.
- The method returns a value of type T.
- The name of the method is `max( )`.
- The method's argument is a Collection. The element type of the collection is specified with a bounded wildcard. We don't know the exact type of the collection's elements, but we know that they have an upper bound of `T`. That is, we know that the elements of the collection are type `T` or a subclass of `T`. Any element of the collection could therefore be used as the return value of the method.

That much is relatively straightforward. We've seen upper-bounded wildcards elsewhere in this section. Now let's look again at the type variable declaration used by the `max( )` method:

```
<T extends Comparable<? super T>>
```

This says first that the type `T` must implement `Comparable`. (Generics syntax uses the keyword `extends` for all type variable bounds, whether classes or interfaces.) This is expected since the purpose of the method is to find the "maximum" object in a collection. But look at the parameterization of the `Comparable` interface. This is a wildcard, but it is bounded with the keyword `super` instead of the keyword `extends`. This is a lower-bounded wildcard. `? extends T` is the familiar upper bound: it means `T` or a subclass. `? super T` is less commonly used: it means `T` or a superclass.

To summarize, then, the type variable declaration states "`T` is a type that is comparable to itself or to some superclass of itself." The `Collections.min()` and `Collections.binarySearch( )` methods have similar signatures.

For other examples of lower-bounded wildcards (that have nothing to do with `Comparable`), consider the `addAll()`, `copy( )`, and `fill()` methods of `Collections`. Here is the signature for `addAll()`:

```
public static <T> boolean addAll(Collection<? super T> c, T... a)
```

This is a varargs method that accepts any number of arguments of type `T` and passes them as a `T[ ]` named *a*. It adds all the elements of *a* to the collection *c*. The element type of the collection is unknown but has a lower bound: the elements are all of type `T` or a superclass of `T`. Whatever the type is, we are assured that the elements of the array are instances of that type, and so it is always legal to add those array elements to the collection.

Recall from our earlier discussion of upper-bounded wildcards that if you have a collection whose element type is an upper-bounded wildcard, it is effectively read-only. Consider `List<? extends Serializable>`. We know that all elements are `Serializable`, so methods like `get()` return a value of type `Serializable`. The compiler won't let us call methods like `add()` because the actual element type of the list is unknown. You can't add arbitrary serializable objects to the list because their implementing class may not be of the correct type.

Since upper-bounded wildcards result in read-only collections, you might expect lower-bounded wildcards to result in write-only collections. This isn't actually the case, however. Suppose we have a `List<? super Integer>`. The actual element type is unknown, but the only possibilities are `Integer` or its ancestors `Number` and `Object`. Whatever the actual type is, it is safe to add `Integer` objects (but not `Number` or `Object` objects) to the list. And, whatever the actual element type is, all elements of the list are instances of `Object`, so `List` methods like `get( )` return `Object` in this case.

Finally, let's turn our attention to the `java.lang.Enum` class. `Enum` serves as the supertype of all enumerated types (described later). It implements the `Comparable` interface but has a confusing generic signature:

```
public class Enum<E extends Enum<E>> implements Comparable<E>, Serializable
```

At first glance, the declaration of the type variable `E` appears circular. Take a closer look though: what this signature really says is that `Enum` must be parameterized by a type that is itself an `Enum`. The reason for this seemingly circular type variable declaration becomes apparent if we look at the `implements` clause of the signature. As we've seen, `Comparable` classes are usually defined to be comparable to themselves. And subclasses of those classes are comparable to their superclass instead. `Enum`, on the other hand, implements the `Comparable` interface not for itself but for a subclass `E`of itself!

Footnotes

> [4] The bound shown here requires that the value type `V` is comparable to itself, in other words, that it implements the `Comparable` interface directly. This rules out the use of types that inherit the `Comparable` interface from a superclass. We'll consider the `Comparable` interface in much more detail at the end of this section and present an alternative there.

View catalog information for Java in a Nutshell, 5th Edition

Return to ONJava.com.