# Generic Methods

Thus far in this chapter, we've talked about generic types and the implementation of generic classes. Now, we're going to look at a different kind of generic animal: *generic methods*. Generic methods essentially do for individual methods what type parameters do for generic classes. But as we'll see, generic methods are smarter and can figure out their parameter types from their usage context without having to be explicitly parameterized. (In reality, of course, it is the compiler that does this.) Generic methods can appear in any class (not just generic classes) and are very useful for a wide variety of applications.

First, let's quickly review the way that we've seen regular methods interact with generic types. We've seen that generic classes can contain methods that use type variables in their arguments and return types in order to adapt themselves to the parameterization of the class. We've also mentioned that generic types themselves can be used in most of the places that any other type can be used. So methods of generic or nongeneric classes can use generic types as argument and return types as well. Here are examples of those usages:

```
// Not generic methods

class GenericClass< T > {
     // method using generic class parameter type
    public void T cache( T entry ) { ... }
}
```

```
    // method using wildcard generic type
    public List<?> reverse( List<?> dates ) { ... }
}
```

The `cache()` method in `GenericClass` accepts an argument of the parameter type `T` and also returns a value of type `T`. The `sortDates()` method, which appears in the nongeneric example class, works with a concrete generic type, and the `reverse()` method works with a wildcard instantiation of a generic type. These are examples of methods that work with generics, but they are not true generic methods.

## Generic Methods Introduced

Like generic classes, generic methods have a parameter type declaration using the <> syntax. This syntax appears before the return type of the method:

```
// generic method
<T> T cache( T entry ) { ... }
```

This `cache()` method looks very much like our earlier example, except that it has its own parameter type declaration that defines the type variable `T`. This method is a generic method and can appear in either a generic or nongeneric class. The scope of `T` is limited to the method `cache()` and hides any definition of `T` in any enclosing generic class. As with generic classes, the type `T` can have bounds:

```
<T extends Entry & Cacheable > T cache( T entry ) { ... }
```

Unlike a generic class, it does not have to be instantiated with a specific parameter type for `T` before it is used. Instead, it *infers* the parameter type `T` from the type of its argument, `entry`. For example:

```
BlogEntry oldEntry = cache( newBlogEntry );
NewspaperEntry old = cache( newNewspaperEntry );
```

Here, our generic method `cache()` inferred the type `BlogEntry` (which we'll presume for the sake of the example is a type of `Entry` and `Cacheable`). `BlogEntry` became the type `T` of the return type and may have been used elsewhere internally by the method. In the next case, the `cache()` method was used on a different type of `Entry` and was able to return the new type in exactly the same way. That's what's powerful about generic methods: the ability to infer a parameter type from their usage context. We'll go into detail about that next.

Another difference with generic class components is that generic methods may be static:

```
class MathUtils {
    public static <T extends Number> T max( T x, T y ) { ... }
}
```

Constructors for classes are essentially methods, too, and follow the same rules as generic methods, minus the return type.

## Type Inference from Arguments

In the previous section, we saw a method infer its type from an argument:

```
<T> T cache( T entry ) { ... }
```

But what if there is more than one argument? We saw just that situation in our last snippet, the static generic method `max( x, y )`. All looks well when we give it two identical types:

```
Integer max = MathUtils.max( new Integer(1), new Integer( 2 ) ) ;
```

```
MathUtils.max( new Integer(1), new Float( 2 ) ) ;
```

In this case, the Java compiler does something really smart. It climbs up the argument type parent classes, looking for the *nearest common supertype*. Java also identifies the *nearest common interfaces* implemented by both of the types. It identifies that both the `Integer` and the `Float` types are subtypes of the `Number` type. It also recognizes that each of these implements (a certain generic instantiation of) the `Comparable` interface. Java then effectively makes this combination of types the parameter type of `T` for this method invocation. The resulting type is, to use the syntax of bounds, `Number & Comparable`. What this means to us is that the result type `T` is assignable to anything matching that particular combination of types.

```
Number max = MathUtils.max( new Integer(1), new Float( 2 ) );
Comparable max = MathUtils.max( new Integer(1), new Float( 2 ) );
```

In English, this statement says that we can work with our `Integer` and our `Float` at the same time only if we think of them as `Numbers` or `Comparables`, which makes sense. The return type has become a new type, which is effectively a `Number` that also implements the `Comparable` interface.

This same inference logic works with any number of arguments. But to be useful, the arguments really have to share some important common supertype or interface. If they don't have anything in common, the result will be their de facto common ancestor, the `Object` type. For example, the nearest common supertype of a `String` and a `List` is `Object` along with the `Serializeable` interface. There's not much a method could do with a type lacking real bounds anyway.

## Type Inference from Assignment Context

We've seen a generic method infer its parameter type from its argument types. But what if the type variable isn't used in any of the arguments or the method has

```
<T> T foo() { ... }
```

You might guess that this is an error because the compiler would appear to have no way of determining what type we want. But it's not! The Java compiler is smart enough to look at the context in which the method is called. Specifically, if the result of the method is assigned to a variable, the compiler tries to make the type of that variable the parameter type. Here's an example. We'll make a factory for our `Trap` objects:

```
<T> Trap<T> makeTrap() { return new Trap<T>(); }

// usage
Trap<Mouse> mouseTrap = makeTrap();
Trap<Bear> bearTrap = makeTrap();
```

The compiler has, as if by magic, determined what kind of instantiation of `Trap` we want based on the assignment context.

Before you get too excited about the possibilities, there's not much you can do with a plain type parameter in the body of that method. For example, we can't create instances of any particular concrete type `T`, so this limits the usefulness of factories. About all we can do is the sort of thing shown here, where we create instances of generics parameterized correctly for the context.

Furthermore, the inference only works on assignment to a variable. Java does not try to guess the parameter type based on the context if the method call is used in other ways, such as to produce an argument to a method or as the value of a return statement from a method. In those cases, the inferred type defaults to type `Object`. (See the section for a solution.)

## Explicit Type Invocation

Although it should not be needed often, a syntax does exist for invoking a generic method with specific parameter types. The syntax is a bit awkward and involves a

```
Integer i = MathUtilities.<Integer>max( 42,  42 );
String s = fooObject.<String>foo( "foo" );
String s = this.<String>foo( "foo" );
```

The prefix must be a class or object instance containing the method. One situation where you'd need to use explicit type invocation is if you are calling a generic method that infers its type from the assignment context, but you are not assigning the value to a variable directly. For example, if you wanted to pass the result of our `makeTrap()` method as a parameter to another method, it would otherwise default to `Object`.

## Wildcard Capture

Generic methods can do one more trick for us involving taming wildcard instantiations of generic types. The term *wildcard capture* refers to the fact that generic methods can work with arguments whose type is a wildcard instantiation of a type, just as if the type were known:

```
<T> Set<T> listToSet( List<T> list ) {
    Set<T> set = new HashSet<T>();
    set.addAll( list );
    return set;
}

// usage
List<?> list = new ArrayList<Date>();
Set<?> set = listToSet( list );
```

The result of these examples is that we converted an unknown instantiation of `List` to an unknown instantiation of `Set`. The type variable `T` represents the actual type of the argument, `list`, for purposes of the method body. The wildcard instantiation must match any bounds of the method parameter type. But because we can work with the type variable only through its bounds types, the

methods.

Another way to look at this is that generic methods are a more powerful alternative to methods using wildcard instantiations of types. We'll do a little comparison next.

## Wildcard Types Versus Generic Methods

You'll recall that trying to work with an object through a wildcard instantiation of its generic type limits us to "reading" the object. We cannot "write" types to the object because its parameter type is unknown. In contrast, because generic methods can infer or "capture" an actual type for their arguments, they allow us to do a lot more with broad ranges of types than we could with wildcard instantiations alone.

For example, suppose we wanted to write a utility method that swaps the first two elements of a list. Using wildcards, we'd like to write something like this:

```
// Bad implementation
List<?> swap( List<?> list ) {
    Object tmp = list.get(0);
    list.set( 0, list.get(1) ); // error, can't write
    list.set( 1, tmp ); // error, can't write
    return list;
}
```

But we are not allowed to call the `set()` method of our list because we don't know what type it actually holds. We are really stuck and there isn't much we can do. But the corresponding generic method gives us a real type to hang our hat:

```
<T> List<T> swapGeneric( List<T> list ) {
    T tmp = list.get( 0 );
    list.set( 0, list.get(1) );
    list.set( 1, tmp );
```

Here, we are able to declare a variable of the correct (inferred) type and write using the `set()` methods appropriately. It would seem that generic methods are the only way to go here. But there is a third path. Wildcard capture, as described in the previous section, allows us to delegate our wildcard version of the method to our actual generic method and use it as if the type were inferred, even though it's open-ended:

```
List<?> swap( List<?> list ) {
    return swapGeneric( list ); // delegate to generic form
}
```

Here, we delegated to the generic version.