The Java® Tutorial

A Short Course on the Basics

Sixth Edition

Raymond Gallardo, Scott Hommel, Sowmya Kannan,
Joni Gordon, Sharon Biocca Zakhour

ORACLE

The Java® Tutorial

Sixth Edition

Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array `T[]` that are greater than a specified element `elem`:

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```


The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as `short`, `int`, `double`, `long`, `float`, `byte`, and `char`. You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Here is the resulting code:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type, provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is called an *is a* relationship. Since an Integer *is a* kind of Object, the assignment is allowed. But Integer is also a kind of Number, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

The same is also true with generics. You can perform a generic type invocation, passing Number as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with Number:


```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>. But what does that mean? Are you allowed to pass in Box<Integer> or Box<Double>, as you might expect? The answer is no because Box<Integer> and Box<Double> are not subtypes of Box<Number>. This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.

Note

Given two concrete types A and B (e.g., Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object. For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see the "Wildcards and Subtyping" section.

Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another is determined by the extends and implements clauses.

Using the Collections classes as an example, ArrayList<E> implements List<E>, and List<E> extends Collection<E>. So ArrayList<String> is a subtype of

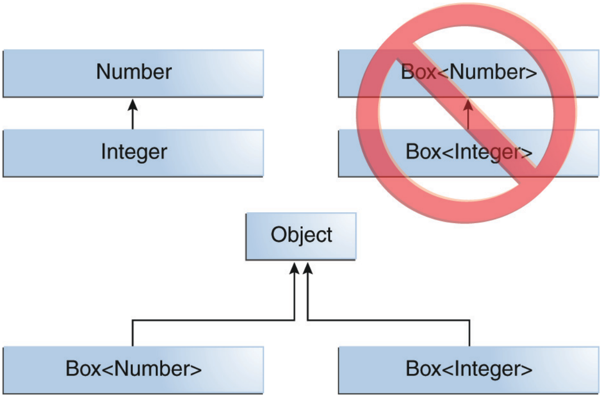

Figure 7.1 Box<Integer> Is Not a Subtype of Box<Number> Even Though Integer Is a Subtype of Number

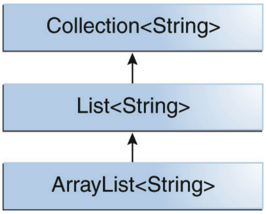

Figure 7.2 A Sample Collections Hierarchy

`List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

Now imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like this:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

The following parameterizations of `PayloadList` are subtypes of `List<String>`:

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`

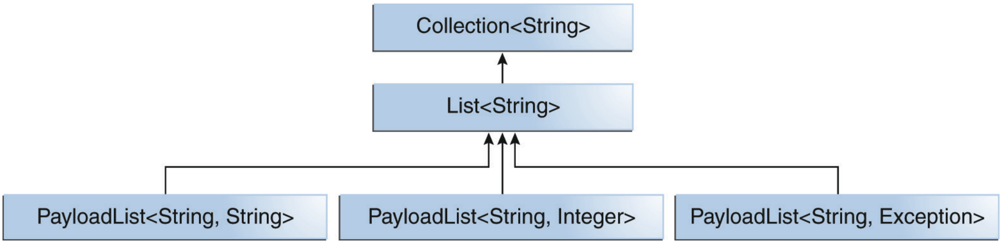

Figure 7.3 A Sample PayloadList Hierarchy

Type Inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that makes the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned or returned. Finally, the inference algorithm tries to find the *most specific* type that works with all the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the `pick` method is of type `Serializable`:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Type Inference and Generic Methods

The previous discussion of generic methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets. Consider the following example, `BoxDemo`, which requires the `Box` class:

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
             boxContents.toString() + "]");
      counter++;
```

```
        }
    }

    public static void main(String[] args) {
        java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
            new java.util.ArrayList<>();
        BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
        BoxDemo.outputBoxes(listOfIntegerBoxes);
    }
}
```

The following is the output from this example:


```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

The generic method `addBox` defines one type parameter named `U`. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them. For example, to invoke the generic method `addBox`, you can specify the type parameter with a *type witness* as follows:

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type witness, a Java compiler automatically infers (from the method's arguments) that the type parameter is `Integer`:

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. As mentioned previously, this pair of angle brackets is informally called *the diamond*. For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the `HashMap()` constructor refers to the `HashMap` raw type, not the `Map<String, List<String>>` type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Type Inference and Generic Constructors of Generic and Nongeneric Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and nongeneric classes. Consider the following example:

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Now consider the following instantiation of the class `MyClass`:

```
new MyClass<Integer>("")
```

This statement creates an instance of the parameterized type `MyClass<Integer>`; the statement explicitly specifies the type `Integer` for the formal type parameter, `X`, of the generic class, `MyClass<X>`. Note that the constructor for this generic class contains a formal type parameter, `T`. The compiler infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class (because the actual parameter of this constructor is a `String` object).

Compilers can infer the actual type parameters of generic constructors, similar to generic methods. However, compilers can also infer the actual type parameters of the generic class being instantiated if you use the diamond (<>). Consider the following example:

```
MyClass<Integer> myObject = new MyClass<>("");
```

In this example, the compiler infers the type `Integer` for the formal type parameter `X` of the generic class `MyClass<X>`. It infers the type `String` for the formal type parameter `T` of the constructor of this generic class.

> **Note**
>
> It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.