Speed Up The Java Development Process

Java Generics and Collections

Maurice Naftalin
& Philip Wadler

O'REILLY®

Table of Contents

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

Chapter 2. Subtyping and Wildcards

Now that we've covered the basics, we can start to cover more-advanced features of generics, such as subtyping and wildcards. In this section, we'll review how subtyping works and we'll see how wildcards let you use subtyping in connection with generics. We'll illustrate our points with examples from the Collections Framework.

2.1. Subtyping and the Substitution Principle

Subtyping is a key feature of object-oriented languages such as Java. In Java, one type is a *subtype* of another if they are related by an `extends` or `implements` clause. Here are some examples:

`Integer`	is a subtype of	`Number`
`Double`	is a subtype of	`Number`
`ArrayList<E>`	is a subtype of	`List<E>`
`List<E>`	is a subtype of	`Collection<E>`
`Collection<E>`	is a subtype of	`Iterable<E>`

Subtyping is transitive, meaning that if one type is a subtype of a second, and the second is a subtype of a third, then the first is a subtype of the third. So, from the last two lines in the preceding list, it follows that `List<E>` is a subtype of `Iterable<E>`. If one type is a subtype of another, we also say that the second is a *supertype* of the first. Every reference type is a subtype of `Object`, and `Object` is a supertype of every reference type. We also say, trivially, that every type is a subtype of itself.

The Substitution Principle tells us that wherever a value of one type is expected, one may provide a value of any subtype of that type:

> *Substitution Principle*: a variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

Consider the interface `Collection<E>`. One of its methods is `add`, which takes a parameter of type `E`:

```
interface Collections<E> {
  public boolean add(E elt);
  ...
}
```


Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756 Prepared for Mitchell Edelman, Safari ID: mitchell.j.edelman@ssa.gov
Publisher: O'Reilly
Print Publication Date: 2006/10/01 User number: 408741 Copyright 2008, Safari Books Online, LLC.

According to the Substitution Principle, if we have a collection of numbers, we may add an integer or a double to it, because `Integer` and `Double` are subtypes of `Number`.

```
List<Number> nums = new ArrayList<Number>();
nums.add(2);
nums.add(3.14);
assert nums.toString().equals("[2, 3.14]");
```

Here, subtyping is used in two ways for each method call. The first call is permitted because `nums` has type `List<Number>`, which is a subtype of `Collection<Number>`, and `2` has type `Integer` (thanks to boxing), which is a subtype of `Number`. The second call is similarly permitted. In both calls, the `E` in `List<E>` is taken to be `Number`.

It may seem reasonable to expect that since `Integer` is a subtype of `Number`, it follows that `List<Integer>` is a subtype of `List<Number>`. But this is *not* the case, because the Substitution Principle would rapidly get us into trouble. It is not always safe to assign a value of type `List<Integer>` to a variable of type `List<Number>`. Consider the following code fragment:

```
List<Integer> ints = Arrays.asList(1,2);
List<Number> nums = ints;  // compile-time error
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

This code assigns variable `ints` to point at a list of integers, and then assigns `nums` to point at the *same* list of integers; hence the call in the third line adds a double to this list, as shown in the fourth line. This must not be allowed! The problem is prevented by observing that here the Substitution Principle does *not* apply: the assignment on the second line is not allowed because `List<Integer>` is not a subtype of `List<Number>`, and the compiler reports that the second line is in error.

What about the reverse? Can we take `List<Number>` to be a subtype of `List<Integer>`? No, that doesn't work either, as shown by the following code:

```
List<Number> nums = Arrays.<Number>asList(2.78, 3.14);
List<Integer> ints = nums;  // compile-time error
assert ints.toString().equals("[2.78, 3.14]");  // uh oh!
```

The problem is prevented by observing that here the Substitution Principle does *not* apply: the assignment on the second line is *not* allowed because `List<Integer>` is *not* a subtype of `List<Number>`, and the compiler reports that the second line is in error.

So `List<Integer>` is not a subtype of `List<Number>`, nor is `List<Number>` a subtype of `List<Integer>`; all we have is the trivial case, where `List<Integer>` is a subtype

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

of itself, and we also have that `List<Integer>` is a subtype of `Collection<Integer>`.

Arrays behave quite differently; with them, `Integer[]` *is* a subtype of `Number[]`. We will compare the treatment of lists and arrays later (see Section 2.5).

Sometimes we would like lists to behave more like arrays, in that we want to accept not only a list with elements of a given type, but also a list with elements of any subtype of a given type. For this purpose, we use *wildcards*.

2.2. Wildcards with extends

Another method in the `Collection` interface is `addAll`, which adds all of the members of one collection to another collection:

```
interface Collection<E> {
  ...
  public boolean addAll(Collection<? extends E> c);
  ...
}
```

Clearly, given a collection of elements of type `E`, it is OK to add all members of another collection with elements of type `E`. The quizzical phrase "`? extends E`" means that it is also OK to add all members of a collection with elements of any type that is a *subtype* of `E`. The question mark is called a *wildcard*, since it stands for some type that is a subtype of `E`.

Here is an example. We create an empty list of numbers, and add to it first a list of integers and then a list of doubles:

```
List<Number> nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1, 2);
List<Double> dbls = Arrays.asList(2.78, 3.14);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.toString().equals("[1, 2, 2.78, 3.14]");
```

The first call is permitted because `nums` has type `List<Number>`, which is a subtype of `Collection<Number>`, and `ints` has type `List<Integer>`, which is a subtype of `Collection<? extends Number>`. The second call is similarly permitted. In both calls, `E` is taken to be `Number`. If the method signature for `addAll` had been written without the wildcard, then the calls to add lists of integers and doubles to a list of numbers would not have been permitted; you would only have been able to add a list that was explicitly declared to be a list of numbers.

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

We can also use wildcards when declaring variables. Here is a variant of the example at the end of the preceding section, changed by adding a wildcard to the second line:

```
List<Integer> ints = Arrays.asList(1,2);
List<? extends Number> nums = ints;
nums.add(3.14);  // compile-time error
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

Before, the second line caused a compile-time error (because `List<Integer>` is not a subtype of `List<Number>`), but the third line was fine (because a double is a number, so you can add a double to a `List<Number>`). Now, the second line is fine (because `List<Integer>` is a subtype of `List<? extends Number>`), but the third line causes a compile-time error (because you cannot add a double to a `List<? extends Number>`, since it might be a list of some other subtype of number). As before, the fourth line shows why one of the preceding lines is illegal!

In general, if a structure contains elements with a type of the form `? extends E`, we can get elements out of the structure, but we cannot put elements into the structure. To put elements into the structure we need another kind of wildcard, as explained in the next section.

2.3. Wildcards with super

Here is a method that copies into a destination list all of the elements from a source list, from the convenience class `Collections`:

```
public static <T> void copy(List<? super T> dst, List<? extends T> src) {
  for (int i = 0; i < src.size(); i++) {
    dst.set(i, src.get(i));
  }
}
```

The quizzical phrase `? super T` means that the destination list may have elements of any type that is a *supertype* of `T`, just as the source list may have elements of any type that is a *subtype* of `T`.

Here is a sample call.

```
List<Object> objs = Arrays.<Object>asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
Collections.copy(objs, ints);
assert objs.toString().equals("[5, 6, four]");
```


Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

As with any generic method, the type parameter may be inferred or may be given explicitly. In this case, there are four possible choices, all of which type-check and all of which have the same effect:

```
Collections.copy(objs, ints);
Collections.<Object>copy(objs, ints);
Collections.<Number>copy(objs, ints);
Collections.<Integer>copy(objs, ints);
```

The first call leaves the type parameter implicit; it is taken to be `Integer`, since that is the most specific choice that works. In the third line, the type parameter `T` is taken to be `Number`. The call is permitted because `objs` has type `List<Object>`, which is a subtype of `List<? super Number>` (since `Object` is a supertype of `Number`, as required by the `super`) and `ints` has type `List<Integer>`, which is a subtype of `List<? extends Number>` (since `Integer` is a subtype of `Number`, as required by the `extends` wildcard).

We could also declare the method with several possible signatures.

```
public static <T> void copy(List<T> dst, List<T> src)
public static <T> void copy(List<T> dst, List<? extends T> src)
public static <T> void copy(List<? super T> dst, List<T> src)
public static <T> void copy(List<? super T> dst, List<? extends T> src)
```

The first of these is too restrictive, as it only permits calls when the destination and source have exactly the same type. The remaining three are equivalent for calls that use implicit type parameters, but differ for explicit type parameters. For the example calls above, the second signature works only when the type parameter is `Object`, the third signature works only when the type parameter is `Integer`, and the last signature works (as we have seen) for all three type parameters—i.e., `Object`, `Number`, and `Integer`. Always use wildcards where you can in a signature, since this permits the widest range of calls.

2.4. The Get and Put Principle

It may be good practice to insert wildcards whenever possible, but how do you decide *which* wildcard to use? Where should you use `extends`, where should you use `super`, and where is it inappropriate to use a wildcard at all?

Fortunately, a simple principle determines which is appropriate.

> *The Get and Put Principle*: use an `extends` wildcard when you only *get* values out of a structure, use a `super` wildcard when you only *put* values into a structure, and don't use a wildcard when you *both* get and put.

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

We already saw this principle at work in the signature of the `copy` method:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

The method gets values out of the source `src`, so it is declared with an `extends` wildcard, and it puts values into the destination `dst`, so it is declared with a `super` wildcard.

Whenever you use an iterator, you get values out of a structure, so use an `extends` wildcard. Here is a method that takes a collection of numbers, converts each to a double, and sums them up:

```
public static double sum(Collection<? extends Number> nums) {
  double s = 0.0;
  for (Number num : nums) s += num.doubleValue();
  return s;
}
```

Since this uses `extends`, all of the following calls are legal:

```
List<Integer> ints = Arrays.asList(1,2,3);
assert sum(ints) == 6.0;

List<Double> doubles = Arrays.asList(2.78,3.14);
assert sum(doubles) == 5.92;

List<Number> nums = Arrays.<Number>asList(1,2,2.78,3.14);
assert sum(nums) == 8.92;
```

The first two calls would not be legal if `extends` was not used.

Whenever you use the `add` method, you put values into a structure, so use a `super` wildcard. Here is a method that takes a collection of numbers and an integer `n`, and puts the first `n` integers, starting from zero, into the collection:

```
public static void count(Collection<? super Integer> ints, int n) {
  for (int i = 0; i < n; i++) ints.add(i);
}
```

Since this uses `super`, all of the following calls are legal:

```
List<Integer> ints = new ArrayList<Integer>();
count(ints, 5);
assert ints.toString().equals("[0, 1, 2, 3, 4]");

List<Number> nums = new ArrayList<Number>();
count(nums, 5);  nums.add(5.0);
assert nums.toString().equals("[0, 1, 2, 3, 4, 5.0]");

List<Object> objs = new ArrayList<Object>();
count(objs, 5);  objs.add("five");
assert objs.toString().equals("[0, 1, 2, 3, 4, five]");
```


Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

The last two calls would not be legal if `super` was not used.

Whenever you both put values into and get values out of the same structure, you should not use a wildcard.

```
public static double sumCount(Collection<Number> nums, int n) {
  count(nums, n);
  return sum(nums);
}
```

The collection is passed to both `sum` and `count`, so its element type must both extend `Number` (as `sum` requires) and be super to `Integer` (as `count` requires). The only two classes that satisfy both of these constraints are `Number` and `Integer`, and we have picked the first of these. Here is a sample call:

```
List<Number> nums = new ArrayList<Number>();
double sum = sumCount(nums,5);
assert sum == 10;
```

Since there is no wildcard, the argument must be a collection of `Number`.

If you don't like having to choose between `Number` and `Integer`, it might occur to you that if Java let you write a wildcard with both `extends` and `super`, you would not need to choose. For instance, we could write the following:

```
double sumCount(Collection<? extends Number super Integer> coll, int n)
// not legal Java!
```

Then we could call `sumCount` on either a collection of numbers or a collection of integers. But Java *doesn't* permit this. The only reason for outlawing it is simplicity, and conceivably Java might support such notation in the future. But, for now, if you need to both get and put then don't use wildcards.

The Get and Put Principle also works the other way around. If an `extends` wildcard is present, pretty much all you will be able to do is get but not put values of that type; and if a `super` wildcard is present, pretty much all you will be able to do is put but not get values of that type.

For example, consider the following code fragment, which uses a list declared with an `extends` wildcard:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
double dbl = sum(nums);  // ok
nums.add(3.14);  // compile-time error
```


Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

The call to `sum` is fine, because it gets values from the list, but the call to `add` is not, because it puts a value into the list. This is just as well, since otherwise we could add a double to a list of integers!

Conversely, consider the following code fragment, which uses a list declared with a `super` wildcard:

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
ints.add(3);  // ok
double dbl = sum(ints);  // compile-time error
```

Now the call to `add` is fine, because it puts a value into the list, but the call to `sum` is not, because it gets a value from the list. This is just as well, because the sum of a list containing a string makes no sense!

The exception proves the rule, and each of these rules has one exception. You cannot put anything into a type declared with an `extends` wildcard—except for the value `null`, which belongs to every reference type:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
nums.add(null);  // ok
assert nums.toString().equals("[1,2,3,null]");
```

Similarly, you cannot get anything out from a type declared with an `extends` wildcard—except for a value of type `Object`, which is a supertype of every reference type:

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```

You may find it helpful to think of `? extends T` as containing every type in an interval bounded by the type of `null` below and by `T` above (where the type of `null` is a subtype of every reference type). Similarly, you may think of `? super T` as containing every type in an interval bounded by `T` below and by `Object` above.

It is tempting to think that an `extends` wildcard ensures immutability, but it does not. As we saw earlier, given a list of type `List<? extends Number>`, you may still add null values to the list. You may also remove list elements (using `remove`, `removeAll`, or `retainAll`) or permute the list (using `swap`, `sort`, or `shuffle` in the convenience class `Collections`; see Section 17.1.1). If you want to ensure that a list cannot be changed, use the method `unmodifiableList` in the class `Collections`; similar methods exist for other collection classes (see Section 17.3.2). If you want to ensure that list elements cannot

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

be changed, consider following the rules for making a class immutable given by Joshua Bloch in his book *Effective Java* (Addison-Wesley) in "Item 13: Favor immutability"; for example, in Part II, the classes `CodingTask` and `PhoneTask` in Section 12.1 are immutable, as is the class `PriorityTask` in Section 13.2.

Because `String` is final and can have no subtypes, you might expect that `List<String>` is the same type as `List<? extends String>`. But in fact the former is a subtype of the latter, but not the same type, as can be seen by an application of our principles. The Substitution Principle tells us it is a subtype, because it is fine to pass a value of the former type where the latter is expected. The Get and Put Principle tells us that it is not the same type, because we can add a string to a value of the former type but not the latter.

2.5. Arrays

It is instructive to compare the treatment of lists and arrays in Java, keeping in mind the Substitution Principle and the Get and Put Principle.

In Java, array subtyping is *covariant*, meaning that type `S[]` is considered to be a subtype of `T[]` whenever `S` is a subtype of `T`. Consider the following code fragment, which allocates an array of integers, assigns it to an array of numbers, and then attempts to assign a double into the array:

```
Integer[] ints = new Integer[] {1,2,3};
Number[] nums = ints;
nums[2] = 3.14;  // array store

 exception


assert Arrays.toString(ints).equals("[1, 2, 3.14]");  // uh oh!
```

Something is wrong with this program, since it puts a double into an array of integers! Where is the problem? Since `Integer[]` is considered a subtype of `Number[]`, according to the Substitution Principle the assignment on the second line must be legal. Instead, the problem is caught on the third line, and it is caught at run time. When an array is allocated (as on the first line), it is tagged with its reified type (a run-time representation of its component type, in this case, `Integer`), and every time an array is assigned into (as on the third line), an array store exception is raised if the reified type is not compatible with the assigned value (in this case, a double cannot be stored into an array of `Integer`).

In contrast, the subtyping relation for generics is *invariant*, meaning that type `List<S>` is *not* considered to be a subtype of `List<T>`, except in the trivial case where

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

`S` and `T` are identical. Here is a code fragment analogous to the preceding one, with lists replacing arrays:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints; // compile-time error
nums.put(2, 3.14);
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

Since `List<Integer>` is not considered to be a subtype of `List<Number>`, the problem is detected on the second line, not the third, and it is detected at compile time, not run time.

Wildcards reintroduce covariant subtyping for generics, in that type `List<S>` *is* considered to be a subtype of `List<? extends T>` when `S` is a subtype of `T`. Here is a third variant of the fragment:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
nums.put(2, 3.14);  // compile-time error
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

As with arrays, the third line is in error, but, in contrast to arrays, the problem is detected at compile time, not run time. The assignment violates the Get and Put Principle, because you cannot put a value into a type declared with an `extends` wildcard.

Wildcards also introduce *contravariant* subtyping for generics, in that type `List<S>` is considered to be a *subtype* of `List<? super T>` when `S` is a *supertype* of `T` (as opposed to a subtype). Arrays do not support contravariant subtyping. For instance, recall that the method `count` accepted a parameter of type `Collection<? super Integer>` and filled it with integers. There is no equivalent way to do this with an array, since Java does not permit you to write `(? super Integer)[]`.

Detecting problems at compile time rather than at run time brings two advantages, one minor and one major. The minor advantage is that it is more efficient. The system does not need to carry around a description of the element type at run time, and the system does not need to check against this description every time an assignment into an array is performed. The major advantage is that a common family of errors is detected by the compiler. This improves every aspect of the program's life cycle: coding, debugging, testing, and maintenance are all made easier, quicker, and less expensive.

Apart from the fact that errors are caught earlier, there are many other reasons to prefer collection classes to arrays. Collections are far more flexible than arrays. The only operations supported on arrays are to get or set a component, and the representation is fixed. Collections support many additional operations, including testing for containment, adding and removing elements, comparing or combining two collections, and extracting

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

a sublist of a list. Collections may be either lists (where order is significant and elements may be repeated) or sets (where order is not significant and elements may not be repeated), and a number of representations are available, including arrays, linked lists, trees, and hash tables. Finally, a comparison of the convenience classes `Collections` and `Arrays` shows that collections offer many operations not provided by arrays, including operations to rotate or shuffle a list, to find the maximum of a collection, and to make a collection unmodifiable or synchronized.

Nonetheless, there are a few cases where arrays are preferred over collections. Arrays of primitive type are much more efficient since they don't involve boxing; and assignments into such an array need not check for an array store exception, because primitive types don't have subtypes. And despite the check for array store exceptions, even arrays of reference type may be more efficient than collection classes with the current generation of compilers, so you may want to use arrays in crucial inner loops. As always, you should measure performance to justify such a design, especially since future compilers may optimize collection classes specially. Finally, in some cases arrays may be preferable for reasons of compatibility.

To summarize, it is better to detect errors at compile time rather than run time, but Java arrays are forced to detect certain errors at run time by the decision to make array subtyping covariant. Was this a good decision? Before the advent of generics, it was absolutely necessary. For instance, look at the following methods, which are used to sort any array or to fill an array with a given value:

```
public static void sort(Object[] a);
public static void fill(Object[] a, Object val);
```

Thanks to covariance, these methods can be used to sort or fill arrays of any reference type. Without covariance and without generics, there would be no way to declare methods that apply for all types. However, now that we have generics, covariant arrays are no longer necessary. Now we can give the methods the following signatures, directly stating that they work for all types:

```
public static <T> void sort(T[] a);
public static <T> void fill(T[] a, T val);
```

In some sense, covariant arrays are an artifact of the lack of generics in earlier versions of Java. Once you have generics, covariant arrays are probably the wrong design choice, and the only reason for retaining them is backward compatibility.

Sections 6.4–6.8 discuss inconvenient interactions between generics and arrays. For many purposes, it may be sensible to consider arrays a deprecated type. We return to this point in Section 6.9.

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

2.6. Wildcards Versus Type Parameters

The `contains` method checks whether a collection contains a given object, and its generalization, `containsAll`, checks whether a collection contains every element of another collection. This section presents two alternate approaches to giving generic signatures for these methods. The first approach uses wildcards and is the one used in the Java Collections Framework. The second approach uses type parameters and is often a more appropriate alternative.

Wildcards Here are the types that the methods have in Java with generics:

```
interface Collection<E> {
  ...
  public boolean contains(Object o);
  public boolean containsAll(Collection<?> c);
  ...
}
```

The first method does not use generics at all! The second method is our first sight of an important abbreviation. The type `Collection<?>` stands for:

```
Collection<? extends Object>
```

Extending `Object` is one of the most common uses of wildcards, so it makes sense to provide a short form for writing it.

These methods let us test for membership and containment:

```
Object obj = "one";
List<Object> objs = Arrays.<Object>asList("one", 2, 3.14, 4);
List<Integer> ints = Arrays.asList(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert !ints.contains(obj);
assert !ints.containsAll(objs);
```

The given list of objects contains both the string `"one"` and the given list of integers, but the given list of integers does not contain the string `"one"`, nor does it contain the given list of objects.

The tests `ints.contains(obj)` and `ints.containsAll(objs)` might seem silly. Of course, a list of integers won't contain an arbitrary object, such as the string `"one"`. But it is permitted because sometimes such tests might succeed:

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756 Prepared for Mitchell Edelman, Safari ID: mitchell.j.edelman@ssa.gov
Publisher: O'Reilly
Print Publication Date: 2006/10/01 User number: 408741 Copyright 2008, Safari Books Online, LLC.

```
Object obj = 1;
List<Object> objs = Arrays.<Object>asList(1, 3);
List<Integer> ints = Arrays.asList(1, 2, 3, 4);
assert ints.contains(obj);
assert ints.containsAll(objs);
```

In this case, the object may be contained in the list of integers because it happens to be an integer, and the list of objects may be contained within the list of integers because every object in the list happens to be an integer.

Type Parameters You might reasonably choose an alternative design for collections—a design in which you can only test containment for subtypes of the element type:

```
interface MyCollection<E> {  // alternative design
  ...
  public boolean contains(E o);
  public boolean containsAll(Collection<? extends E> c);
  ...
}
```

Say we have a class `MyList` that implements `MyCollection`. Now the tests are legal only one way around:

```
Object obj = "one";
MyList<Object> objs = MyList.<Object>asList("one", 2, 3.14, 4);
MyList<Integer> ints = MyList.asList(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints)
assert !ints.contains(obj);       // compile-time error
assert !ints.containsAll(objs);   // compile-time error
```

The last two tests are illegal, because the type declarations require that we can only test whether a list contains an element of a subtype of that list. So we can check whether a list of objects contains a list of integers, but not the other way around.

Which of the two styles is better is a matter of taste. The first permits more tests, and the second catches more errors at compile time (while also ruling out some sensible tests). The designers of the Java libraries chose the first, more liberal, alternative, because someone using the Collections Framework *before* generics might well have written a test such as `ints.containsAll(objs)`, and that person would want that test to remain valid *after* generics were added to Java. However, when designing a new generic library, such as `MyCollection`, when backward compatibility is less important, the design that catches more errors at compile time might make more sense.

Arguably, the library designers made the wrong choice. Only rarely will a test such as `ints.containsAll(objs)` be required, and such a test can still be permitted by declaring `ints` to have type `List<Object>` rather than type `List<Integer>`. It might

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756 Prepared for Mitchell Edelman, Safari ID: mitchell.j.edelman@ssa.gov
Publisher: O'Reilly
Print Publication Date: 2006/10/01 User number: 408741 Copyright 2008, Safari Books Online, LLC.

have been better to catch more errors in the common case rather than to permit more-precise typing in an uncommon case.

The same design choice applies to other methods that contain `Object` or `Collection<?>` in their signature, such as `remove`, `removeAll`, and `retainAll`.

2.7. Wildcard Capture

When a generic method is invoked, the type parameter may be chosen to match the unknown type represented by a wildcard. This is called *wildcard capture*.

Consider the method `reverse` in the convenience class `java.util.Collections`, which accepts a list of any type and reverses it. It can be given either of the following two signatures, which are equivalent:

```
public static void reverse(List<?> list);
public static void <T> reverse(List<T> list);
```

The wildcard signature is slightly shorter and clearer, and is the one used in the library.

If you use the second signature, it is easy to implement the method:

```
public static void <T> reverse(List<T> list) {
  List<T> tmp = new ArrayList<T>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1));
  }
}
```

This copies the argument into a temporary list, and then writes from the copy back into the original in reverse order.

If you try to use the first signature with a similar method body, it won't work:

```
public static void reverse(List<?> list) {
  List<Object> tmp = new ArrayList<Object>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1));  // compile-time error
  }
}
```

Now it is not legal to write from the copy back into the original, because we are trying to write from a list of objects into a list of unknown type. Replacing `List<Object>` with `List<?>` won't fix the problem, because now we have two lists with (possibly different) unknown element types.

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

Instead, you can implement the method with the first signature by implementing a private method with the second signature, and calling the second from the first:

```
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
  List<T> tmp = new ArrayList<T>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1));
  }
}
```

Here we say that the type variable `T` has *captured* the wildcard. This is a generally useful technique when dealing with wildcards, and it is worth knowing.

Another reason to know about wildcard capture is that it can show up in error messages, even if you don't use the above technique. In general, each occurrence of a wildcard is taken to stand for some unknown type. If the compiler prints an error message containing this type, it is referred to as `capture of ?`. For instance, with Sun's current compiler, the incorrect version of `reverse` generates the following error message:

```
Capture.java:6: set(int,capture of ?) in java.util.List<capture of ?>
cannot be applied to (int,java.lang.Object)
            list.set(i, tmp.get(list.size()-i-1));
                ^
```

Hence, if you see the quizzical phrase `capture of ?` in an error message, it will come from a wildcard type. Even if there are two distinct wildcards, the compiler will print the type associated with each as `capture of ?`. Bounded wildcards generate names that are even more long-winded, such as `capture of ? extends Number`.

2.8. Restrictions on Wildcards

Wildcards may not appear at the top level in class instance creation expressions (`new`), in explicit type parameters in generic method calls, or in supertypes (`extends` and `implements`).

Instance Creation In a class instance creation expression, if the type is a parameterized type, then none of the type parameters may be wildcards. For example, the following are illegal:

```
List<?> list = new ArrayList<?>();  // compile-time error
Map<String, ? extends Number> map
  = new HashMap<String, ? extends Number>();  // compile-time error
```


Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

This is usually not a hardship. The Get and Put Principle tells us that if a structure contains a wildcard, we should only get values out of it (if it is an `extends` wildcard) or only put values into it (if it is a `super` wildcard). For a structure to be useful, we must do both. Therefore, we usually create a structure at a precise type, even if we use wildcard types to put values into or get values from the structure, as in the following example:

```
List<Number> nums = new ArrayList<Number>();
List<? super Number> sink = nums;
List<? extends Number> source = nums;
for (int i=0; i<10; i++) sink.add(i);
double sum=0; for (Number num : source) sum+=num.doubleValue();
```

Here wildcards appear in the second and third lines, but not in the first line that creates the list.

Only top-level parameters in instance creation are prohibited from containing wildcards. Nested wildcards are permitted. Hence, the following is legal:

```
List<List<?>> lists = new ArrayList<List<?>>();
lists.add(Arrays.asList(1,2,3));
lists.add(Arrays.asList("four","five"));
assert lists.toString().equals("[[1, 2, 3], [four, five]]");
```

Even though the list of lists is created at a wildcard type, each individual list within it has a specific type: the first is a list of integers and the second is a list of strings. The wildcard type prohibits us from extracting elements from the inner lists at any type other than `Object`, but since that is the type used by `toString`, this code is well typed.

One way to remember the restriction is that the relationship between wildcards and ordinary types is similar to the relationship between interfaces and classes—wildcards and interfaces are more general, ordinary types and classes are more specific, and instance creation requires the more specific information. Consider the following three statements:

```
List<?> list = new ArrayList<Object>();   // ok
List<?> list = new List<Object>()  // compile-time error
List<?> list = new ArrayList<?>()  // compile-time error
```

The first is legal; the second is illegal because an instance creation expression requires a class, not an interface; and the third is illegal because an instance creation expression requires an ordinary type, not a wildcard.

You might wonder why this restriction is necessary. The Java designers had in mind that every wildcard type is shorthand for some ordinary type, so they believed that ultimately every object should be created with an ordinary type. It is not clear whether this restriction is necessary, but it is unlikely to be a problem. (We tried hard to contrive a situation in which it was a problem, and we failed!)

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01

Generic Method Calls If a generic method call includes explicit type parameters, those type parameters must not be wildcards. For example, say we have the following generic method:

```
class Lists {
  public static <T> List<T> factory() { return new ArrayList<T>(); }
}
```

You may choose for the type parameters to be inferred, or you may pass an explicit type parameter. Both of the following are legal:

```
List<?> list = Lists.factory();
List<?> list = Lists.<Object>factory();
```

If an explicit type parameter is passed, it must not be a wildcard:

```
List<?> list = Lists.<?>factory();  // compile-time error
```

As before, nested wildcards are permitted:

```
List<List<?>> = Lists.<List<?>>factory();  // ok
```

The motivation for this restriction is similar to the previous one. Again, it is not clear whether it is necessary, but it is unlikely to be a problem.

Supertypes When a class instance is created, it invokes the initializer for its supertype. Hence, any restriction that applies to instance creation must also apply to supertypes. In a class declaration, if the supertype or any superinterface has type parameters, these types must not be wildcards.

For example, this declaration is illegal:

```
class AnyList extends ArrayList<?> {...} // compile-time error
```

And so is this:

```
class AnotherList implements List<?> {...} // compile-time error
```

But, as before, nested wildcards are permitted:

```
class NestedList implements ArrayList<List<?>> {...}  // ok
```

The motivation for this restriction is similar to the previous two. As before, it is not clear whether it is necessary, but it is unlikely to be a problem.

Java Generics and Collections By Philip Wadler, Maurice Naftalin ISBN: 0596527756
Publisher: O'Reilly
Print Publication Date: 2006/10/01