Subtyping and Wildcards

18 September 2025 20:18

Subtyping and the Substitution Principle

Subtyping is a key feature of object oriented languages such as Java. In Java one type is subtype of another if they are related by extents 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 2nd, and the 2nd is a subtype of 3rd, then the 1st is a subtype of the 3rd.
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 define the subtype relationship so that every type is a subtype of itself.

Substitution Principle: Wherever a value of type T is expected. You can provide instead of value of a subtype of T.
For example, 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 method is add, which takes a parameter of type E:

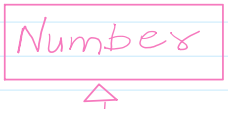
```
interface Collection<E>{
    …
    public boolean add(E element);
    …
}
```

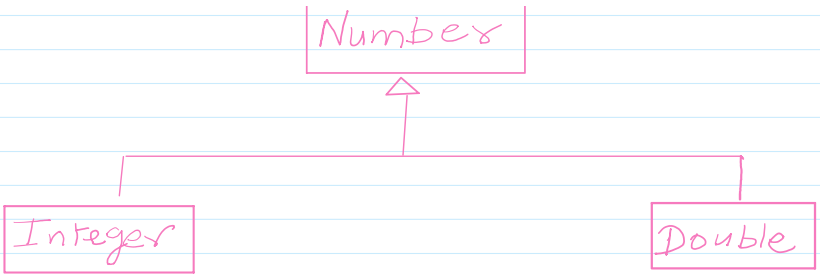
According to the substitution principle, if we have a collection of numbers, we may add an integer or a double to it because Integer and Double are subtypes of Number.

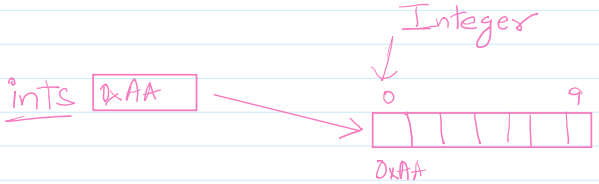
```
jshell> List<Number> nums = new ArrayList<>();
nums ==> []

jshell> nums.add(2)
$3 ==> true

jshell> nums.add(0.25)
$4 ==> true

jshell> nums
nums ==> [2, 0.25]
```

Here subtyping is used in two ways for each method call. The first call is permitted because numbs has a type List<Number> Which is a subtype of Collection<Number> And '2' has a type Integer. 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 if it were, the substitution principle would rapidly get us into trouble. To see why it is not safe to assign a value of type List<Integer> To a variable of type List <Number> Consider the following code:








```
jshell> List<Integer> ints = new ArrayList<>();
ints ==> []

jshell> nums = ints;
|  Error:
|  incompatible types: java.util.List<java.lang.Integer> cannot be converted to
java.util.List<java.lang.Number>
|  nums = ints;

|         ^--^
nums.add(3.14);
```

This code assigns the variable ints to point to a list of Integer and then if the substitution principle where required to succeed. Assigns nums to point to the same list; In that case the call in the following line which is clearly legal, would add double into this list, meaning that the variable ints, typed as a List<Integer> would be pointing at a list containing a Double, and Java's type system would be broken. Obviously this can't be allowed. The solution is to outlaw assignment. If List<Integer> is prevented from being a subtype of List<Number> Then the substitution principle does not apply and can be reported as a compile error.

What about the reverse?

```
jshell> List<Number> nums = new ArrayList<>();
nums ==> []

jshell> nums.add(3.14);
$8 ==> true

jshell> nums
nums ==> [3.14]

jshell> List<Integer> ints = nums;
|  Error:
|  incompatible types: java.util.List<java.lang.Number> cannot be converted to
java.util.List<java.lang.Integer>
|  List<Integer> ints = nums;
|                       ^--^
```

Where allowing leads to the same problem once again, a variable typed as List<Integer> is pointing to a list containing a Double. So List<Number> Is not a subtype of List<Integer> And since we have already seen that List<Integer> is not a subtype of List<Number> And all we have is this trivial case where List<Integer> is a subtype of itself.

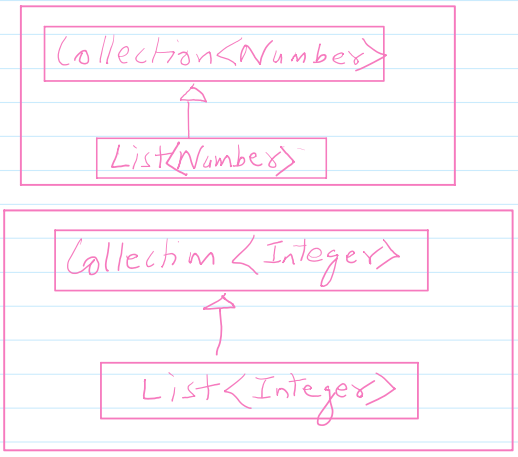
Wildcards with extends

Another method in the Collection interface is addAll, Which adds all members of one collection to another.

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

}

The phrase "? extends E" means that it is also OK to add all members of Collection with elements of any type that is a **subtype** of E. The question mark is called a wild card. It stands for some-as-yet-unknown subtype of E.




```
jshell> import java.util.*;

jshell> List<Number> nums = new ArrayList<>();
nums ==> []

jshell> List<Integer> ints = List.of(1,2,3,4);
ints ==> [1, 2, 3, 4]

jshell> List<Double> dbls = List.of(1.0, 2.0, 0.5, 0.7);
dbls ==> [1.0, 2.0, 0.5, 0.7]

jshell> nums.addAll(ints);
$5 ==> true

jshell> nums
nums ==> [1, 2, 3, 4]

jshell> nums.addAll(dbls);
$7 ==> true

jshell> nums
nums ==> [1, 2, 3, 4, 1.0, 2.0, 0.5, 0.7]
```

The first call is permitted because nums has a type List<Number> Which is a subtype of Collection<Number> and ints has type List<Integer> which is a subtype of Collection<Integer>, Which in turn is a subtype of Collection<? extends Number>. The second call is similarly permitted. In both calls, E is taken to be a number. If the parameter declaration for Add all has been written without the wild card, then the calls to add list of integers and double S to a list of numbers would not have been permitted. You would only have been able to add list which was explicitly declared to be as a list of numbers.

We can also use wildcards when declaring variables.

```
jshell> List<Integer> myint = new ArrayList<>();
myint ==> []

jshell> List<? extends Number> nums = myint;
nums ==> []

jshell> nums.add(3.14);
|  Error:
|  incompatible types: double cannot be converted to capture#2 of ? extends java.lang.Number
|  nums.add(3.14);
```

You cannot add a double to a List<? extends Number>, since you don't know the type represented by the wildcard; all you know that it is some type of Number. Otherwise, as before, the last line would result in the variable myints typed as a List<Integer>, pointing to a list containing a double.

In general, if a structure contains elements with type of the form "? extends E" we can get the element out of the structure, but we cannot put elements into the structure. To put elements in the structure we need another kind of wildcard with super.

Wildcards with super

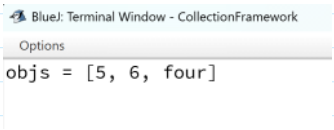
Here is a method from the convenience class Collections that copies the element from a source list into a destination list:

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

      for(int i = 0; i < src.size(); i++){
         dst.set(i, src.get(i));
      }
}
```

The quizzical phrase "? super T" means 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.

```
package WildCardsAndSubtyping;
import java.util.*;
import java.util.stream.*;
import java.io.*;
/**
 * Write a description of class TestWildCards here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class TestWildCards
{
    public static void main(String[] args){
        List<Object> objs = Stream.of(2, 3.14, "four").collect(Collectors.toList());
        List<Integer> ints = List.of(5, 6);
        Collections.copy(objs, ints);
        System.out.print("\fobjs = " + objs);
    }
}
```

Output:



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 effects.

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

The first call leaves the parameter implicit. It is taken to be integer since that is the most specific choice that works. In the 3rd line it is taken to be Number. The call is permitted because objs has type List<Object>, which is the subtype of List<? super Number> (Since Object is a supertype of Number as required by super wild card) and ints has type List<Integer> Which is a super type of. List<? extends Number>(Since Integer is a subtype of Number as required by the extents wild card.)

We could also declare the method with several possible signatures. (A method signature is a combination of method identifier, its type parameters, and the types and order of its parameters).

```
 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 two 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. For all three type parameters, that is Object, Number and Integer. When writing a method signature, always use wildcards where you can, since this permits the widest range of calls.

The Get and Put Principle

It may be good practice to insert wildcards whenever possible, but how do you decide which wild card to use? Where should you use extends and where should you use super and where it is inappropriate to use a wild card 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 the structure, use a **super** wildcard when you only put values into a structure, and don't use a wild card when you both get and put.*

In Effective Java, Joshua Bloch gives this principle the mnemonic PECS(producer extends, consumer super). We already saw this principle at work in the signature of the copy method.

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

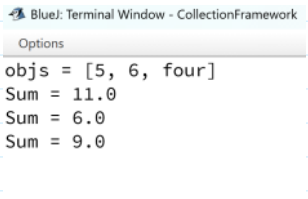
Whenever you use an iterator or a stream, you are getting values out of a structure, so you must use an extends wildcard. Here is the iterator version of a method that takes a collection of numbers and convert each to double and sums them up:

```
package WildCardsAndSubtyping;
import java.util.*;
import java.util.stream.*;
import java.io.*;
/**
 * Write a description of class TestWildCards here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class TestWildCards
{
    public static void main(String[] args){
        List<Object> objs = Stream.of(2, 3.14, "four").collect(Collectors.toList());
        List<Integer> ints = List.of(5, 6);
        //Collections.copy(objs, ints);
        //Collections.<Object>copy(objs, ints);
        //Collections.<Number>copy(objs, ints);
        Collections.<Integer>copy(objs, ints);
        System.out.print("\fobjs = " + objs);
        System.out.print("\nSum = " + sum(ints));

        List<Double> doubles = List.of(2.5, 3.5);
        System.out.print("\nSum = " + sum(doubles));
        List<Number> nums = List.of(1, 2, 2.5,3.5);
        System.out.print("\nSum = " + sum(nums));
    }

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

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

Since, this method uses extends, the above calls are legal.

```
        return s;
        */
        return nums.stream().mapToDouble(Number::doubleValue).sum();
    }
}
```



Whenever you use the add method you are putting values into a structure, so you should use a super wild card. Here is a method that takes a collection of number and an integer N and puts the first N integers starting from 0 into the collection.

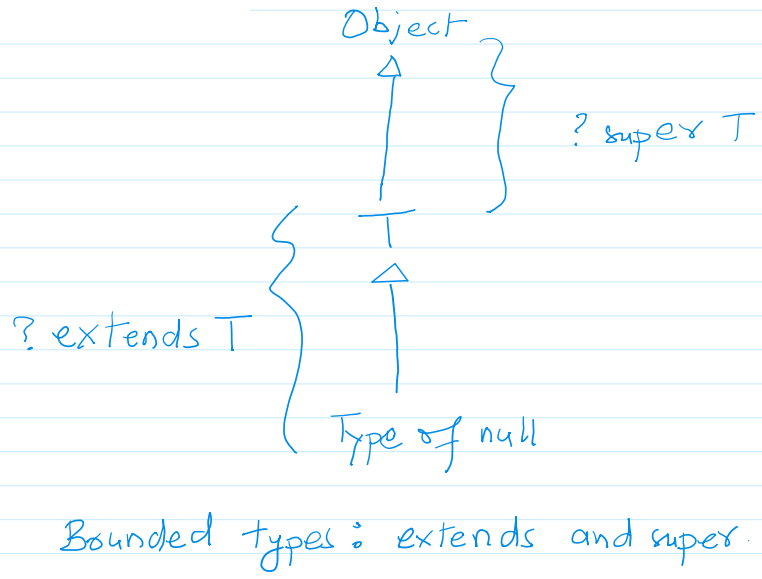
```
package WildCardsAndSubtyping;
import java.util.*;
import java.util.stream.*;
import java.io.*;
/**
 * Write a description of class TestWildCards here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class TestWildCards
{
    public static void main(String[] args){
        List<Object> objs = Stream.of(2, 3.14, "four").collect(Collectors.toList());
        List<Integer> ints = List.of(5, 6);
        //Collections.copy(objs, ints);
        //Collections.<Object>copy(objs, ints);
        //Collections.<Number>copy(objs, ints);
        Collections.<Integer>copy(objs, ints);
        System.out.print("\fobjs = " + objs);
        System.out.print("\nSum = " + sum(ints));

        List<Double> doubles = List.of(2.5, 3.5);
        System.out.print("\nSum = " + sum(doubles));
        List<Number> nums = List.of(1, 2, 2.5,3.5);
        System.out.print("\nSum = " + sum(nums));

        List<Integer> ints1 = new ArrayList<>();
        add(ints1, 5);
        System.out.print("\nints1 = " + ints1);

        List<Number> nums1 = new ArrayList<>();
        add(nums1, 5);
        nums1.add(5.0);
        System.out.print("\nnums1 = " + nums1);

        List<Object> objs1 = new ArrayList<>();
        add(objs1, 5);
        objs1.add("Five");
        System.out.print("\nobjs1 = " + objs1);
    }

    public static double sum(Collection<? extends Number> nums){
        /*
        double s = 0.0;
        for(Number num : nums){
```

← These calls would be illegal if super was not used

```
        /*
        double s = 0.0;
        for(Number num : nums){
            s += num.doubleValue();
        }
        return s;
        */
        return nums.stream().mapToDouble(Number::doubleValue).sum();
    }

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

Output:

```
BlueJ: Terminal Window - CollectionFramework
Options
objs = [5, 6, four]
Sum = 11.0
Sum = 6.0
Sum = 9.0
ints1 = [0, 1, 2, 3, 4]
nums1 = [0, 1, 2, 3, 4, 5.0]
objs1 = [0, 1, 2, 3, 4, Five]
```



Bounded types: extends and super.

It is tempting to think an extends wildcard ensures immutability or at least un-modifiability, but it does not. As we saw earlier, given a type of List<? extends Number> you can still add null values to the list. You can also remove list elements(using remove, removeAll or retainAll) or permute the list(using swap, sort, or shuffle in the convenience class Collections.

Because String is a 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 latter, not the same type as can be seen by the applications of our principles. The substitution principle tells us it is a subtype because it is fine to pass a value of a former type where the latter is expected. The get and put principal tells us that. It is not the same type because we can add a string value of the former type, but not the latter.

$C = 2\pi r$

$\uparrow 6.28 \times$

tau

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 subtype of T[] whenever S is a subtype of T.

```
import java.util.*;

/**
 * Write a description of class TestWildCardArray here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class TestWildCardArray
{
    public static void main(String[] args){
        Integer[] x = {0};
        Number[] nums = x;
        nums[0] = 6.28;
    }
}
```

java.lang.ArrayStoreException: java.lang.Double

Something is wrong with this program since. If it were to succeed, the variable x, types as Integer[], Would be pointing at an array containing a Double, and once again Java's type system would be broken. Where is the problem? Since integer [] is considered a subtype of Number [] according to the substitution principle, the assignment on the 2nd line must be legal. Instead, the problem is caught at runtime on the 3rd line. Where an array is allocated (as on the first line) as it tagged with its reified type (or runtime representation of its component type, in this case Integer), every time a value is stored in the array(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 Integer array).

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 S and T are identical.

```
List<Integer> ints = Arrays.asList(0);
List<Number> num = ints;
```

incompatible types: java.util.List<java.lang.Integer> cannot be converted to java.util.List<java.lang.Number>

Since List<Integer> Is not considered to be a subtype of List<Number> the problem is detected on the second line. And it is detected at compile time, not run time.
Wildcards reintroduce covariance subtyping for generics in that type List<S> Is considered to be the subtype of List<? extends T> When S is a subtype of T.

```
List<Integer> ints = Arrays.asList(0);
List<? extends Number> num = ints;
num.set(0, 3.14);
```

incompatible types: double cannot be converted to capture#1 of ? extends java.lang.Number

As with arrays. The 3rd line is in error, but in contrast to arrays, the problem is detected at compile time, not runtime. The assignment violates the get and put principal because you cannot put a value into a type declared with an extends wild card.

Wildcards also introduce covariant subtyping for generics in that type List<S> is considered to be a subtype of List<? super T> when S is a super type of T (as opposed to a subtype) Arrays do not support covariant subtyping. For instance, recall that method add(). 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 runtime 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 runtime, 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 all made easier, quicker and less expensive.

Is using Arrays is better or Collections class is better!
Apart from the fact that typing errors are caught earlier, there are many other reasons to prefer collection classes to arrays. Collections are more flexible than arrays. The only operation supported on arrays are to get or set a component, and the number of elements they contain is fixed. Collection supports many additional operations, including testing for containment, adding and removing elements, and combining two collections. Collections maybe list (where order is significant and elements may be repeated and further operations are available), Sets. (where order is not significant and elements may not be repeated), Or queues (predominantly used in workflow situation.) and a number of representations are available, including arrays, linked list, trees, and hash tables. Specialized collections are available for situations requiring efficient concurrent access. Finally, although this is not an inherent advantage of collections over arrays. The convenience class Collections offer operations to rotate or shuffle a list. To find the maximum of a collection, to make a collection unmodifiable or synchronized, and others. Many more than are provided by the corresponding convenience class Arrays.
Nonetheless, there are situation in which arrays are preferable to collections. In particular, arrays of primitive types are much more efficient than either arrays or collections of reference types. Since they don't involve boxing and they use less memory with much better spatial locality. Further assignments into primitive array need not check for array store exception because arrays of primitive type do not have subtypes. These advantages may lead you to replace collection with arrays in performance critical code sections. As always, you should measure comparative performance to justify such a design, bearing in mind that future compiler and library design improvements may change the relative performance balance. 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 runtime, but Java arrays are forced to detect certain errors at runtime 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 is used to sort any array or to fill any 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 salt or fill arrays of any reference type. Without covariance and without Generics, there would have been no way to declare methods that apply for all types. With generics, however, covariant arrays become unnecessary. These methods could now have the following signature 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.

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. The first approach uses wild cards and is the one used in the Java Collections Framework. The second approach uses type parameters.

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

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

Wildcard

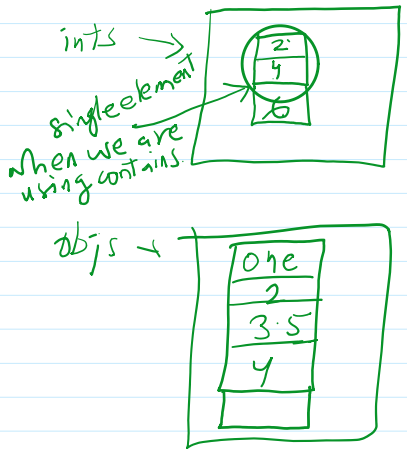
```
    }
```

The first method does not use generics at all. The second method is our first site 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.

```
import java.util.*;
public class WildCardVsTypeParameter
{
    public static void main(String args[]){
        Object obj = "one";
        List<Object> objs = List.of("one", 2, 3.5, 4);
        List<Integer> ints = List.of(2, 4);
        System.out.print("\f" + objs.contains(obj)); // true
        System.out.print("\n" + objs.contains(ints)); // false
        System.out.print("\n" + ints.contains(obj)); // false
        System.out.print("\n" + ints.containsAll(objs)); // false
        System.out.print("\n" + objs.containsAll(ints)); // true
    }
}
```



Output:

```
BlueJ: Terminal Window - CollectionFramework
Options
true
false
false
false
true
```

```
import java.util.*;

public class WildCardVsTypeParameter
{
    public static void main(String args[]){
        Object obj = 1;
        List<Object> objs = List.of(1, 3);
        List<Integer> ints = List.of(1, 2, 3, 4);
        System.out.print("\f" + objs.contains(obj));
        System.out.print("\n" + objs.contains(ints));//false
        System.out.print("\n" + ints.contains(obj));//true
        System.out.print("\n" + ints.containsAll(objs));//true
        System.out.print("\n" + objs.containsAll(ints));//false
    }
}
```

Output:

```
BlueJ: Terminal Window - CollectionFramework
Options
true
false
true
true
false
```

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

the list happens to be an integer.

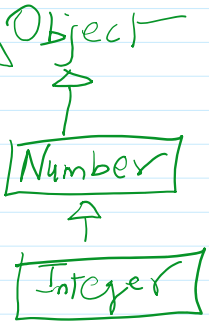
Type Parameters: You might reasonably choose an alternative design for collections. A design in which you can only test containment for subtypes of the element type:

Type parameter

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

Type Parameter

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




```
import java.util.*;
public class WildCardVsTypeParameter
{
    public static void main(String args[]){
        Object obj = "one";
        List<Object> objs = List.of("one", 2, 3.5, 4);
        List<Integer> ints = List.of(2, 4);
        System.out.print("\f" + objs.contains(obj));
        System.out.print("\n" + objs.contains(ints));
        System.out.print("\n" + objs.containsAll(ints));
        System.out.print("\n" + ints.contains(obj));  //compile-time error
        System.out.print("\n" + ints.containsAll(objs)); //compile-time error
    }
}
```

String Double Integer

The last two tests are illegal, because the type declaration now 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.

The library designers chose wildcards over type parameter in case of contains, containsAll, and other methods in Collection, Lists, and Map. that test for, or remove values from, the collection..

Wildcard Capture(Skill)

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 accept 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 <T> void reverse(List<T> list);
```

The wild card 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<T> void reverse(List<T> list){
    List<T> tmp = new ArrayList<>(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 copy back into the original in reverse order.

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

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

incompatible types: java.lang.Object cannot be converted to capture#1 of ?

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.

Instead, you can implement the method with the first signature by implementing a private method with the 2nd signature and calling the 2nd from the 1st.

```
public class WildCardCapture
{
    //2nd method
    private static<T> void rev(List<T> list){
        List<T> tmp = new ArrayList<>(list);
        for(int i = 0; i < list.size(); i++){
            list.set(i, tmp.get(list.size() - i - 1));
        }
    }

    public static void reverse(List<?> list){
        rev(list);
    }
}
```

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

Restrictions on Wildcards
Wild cards may not appear at the top level in class **instance creation expression** (new) in explicit type parameters, in **generic method calls**, or in a **super types** (extends and implements).

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

```
List<?> list1 = new ArrayList<?>();
```

unexpected type
required: class or interface without bounds
found: ?

← compile-time error

```
Map<String, ? extends Number> map = new HashMap<String, ? extends Number>();
```

unexpected type
required: class or interface without bounds
found: ? extends java.lang.Number

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

```
public static void restrictWC(){
  1 List<Number> nums = new ArrayList<Number>();
  2 List<? super Number> sink = nums;
  3 List<? extends Number> source = nums;
    for(int i = 0; i < 4; i++) sink.add(i);
```

← instantion, no wild cards are used.

```
        for(int i = 0; i < 4; i++) sink.add(i);
        int sum = nums.stream().mapToInt(Number::intValue).sum();
        System.out.print("\nSum = " + sum);
    }
```

abstract method in Number class,

The job of this method is to convert each element of IntStream sequence into 'int' type,

Stream<Number> stream()

Returns a sequential `Stream` with this collection as its source.

This method should be overridden when the {@link #spliterator()} method cannot return a spliterator that is `IMMUTABLE`, `CONCURRENT`, or *late-binding*. (See {@link #spliterator()} for details.)

implSpec - The default implementation creates a sequential `Stream` from the collection's `Spliterator`.
return - a sequential `Stream` over the elements in this collection
since - 1.8

IntStream mapToInt(ToIntFunction<? super Number> mapper)

Returns an `IntStream` consisting of the results of applying the given function to the elements of this stream.

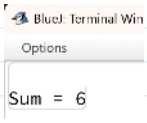
This is an intermediate operation.

Parameters

mapper - a non-interfering, stateless function to apply to each element

return - the new stream

Output:



intValue

```
public int intValue()
```

Returns the value of this Integer as an int.

Specified by:
intValue in class Number
Returns:
the numeric value represented by this object after conversion to type int.

Here wild card appears in the second and third lines but not in the first line that creates the list. Only top level parameters in instance creations are prohibited from containing wild card. Nested wild cards are permitted. Hence the following is legal:

```
    public static void restrictWC1(){
        List<List<?>> lists = new ArrayList<List<?>>();
        lists.add(List.of(1,2,3));
        lists.add(List.of("four", "five"));
        System.out.print("\n"+ lists);
    }
```

Output:

```
[[1, 2, 3], [four, five]]
```

Even though the list of list is created at a wild card type, each individual list within it has a specific type. The first is the list of integer and the 2nd is a list of strings. The wild card type prohibits us from extracting elements from the inner list as 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. Wild cards and interfaces are more general. Ordinary types and classes are more specific, and an instance creation requires more specific information.

```
    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 3rd is illegal because an instance creation expression requires an ordinary type, not a wild card.

My Opinion:
You might wonder why this restriction is necessary. The Java designers had in mind that every wild card 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. (I tried hard to contrive a situation in which it was a problem and I failed)

Generic Method Calls: If a generic method call includes explicit type parameters, those type parameters must not be wild cards. 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 parameter to be inferred, or you may pass an explicit type parameter. Both of the following are legal:

```
    List<?> list = Lists.factory();
             OR
    List<?> list = Lists.<Object>factory();
                                ← wild card is not allowed here
    List<?> list2 = Lists.<?>factory(); //compile-time error
```

As before, nested wildcards are permitted:

```
  List<List<?>> list2 = Lists.<List<?>>factory(); //OK
```

Opinion
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.

Super types: When a class instance is created, it invokes the constructor 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 super interface has type parameters, these types must not be wild cards. For example, this declaration is illegal:

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

And so is this:

```
  class OtherList implements List<?>{ //compile-time error
     …
  }
```

But, as before, nested wildcards are permitted:

```
  class NestedList extends ArrayList<List<?>>{   //OK
  …
  }
```