Wildcards

Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on `List<Integer>`, `List<Double>`, and `List<Number>`; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('`?`'), followed by the `extends` keyword, followed by its upper bound. Note that, in this context, `extends` is used in a general sense to mean either "`extends`" (as in classes) or "`implements`" (as in interfaces).

To write the method that works on lists of `Number` and the subtypes of `Number`, such as `Integer`, `Double`, and `Float`, you would specify `List<? extends Number>`. The term `List<Number>` is more restrictive than `List<? extends Number>` because the former matches a list of type `Number` only, whereas the latter matches a list of type `Number` or any of its subclasses.

Consider the following process method:

```java
public static void process(List<? extends Foo> list) { /* ... */ }
```

The upper bounded wildcard, `<? extends Foo>`, where `Foo` is any type, matches `Foo` and any subtype of `Foo`. The process method can access the list elements as type `Foo`:

```java
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

In the `foreach` clause, the `elem` variable iterates over each element in the list. Any method defined in the `Foo` class can now be used on `elem`.

The `sumOfList()` method returns the sum of the numbers in a list:

```java
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

The following code, using a list of `Integer` objects, prints `sum = 6.0`:

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

A list of `Double` values can use the same `sumOfList()` method. The following code prints `sum = 7.0`:

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (`?`), for example, `List<?>`. This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the `Object` class.
- When the code is using methods in the generic class that do not depend on the type parameter. For example, `List.size()` or `List.clear()`. In fact, `Class<?>` is so often used because most of the methods in `Class<T>` do not depend on `T`.

Consider the following method, `printList()`:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of `printList()` is to print a list of any type, but it fails to achieve that goal — it prints only a list of `Object` instances; it cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`. To write a generic `printList()` method, use `List<?>`:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Because for any concrete type `A`, `List<A>` is a subtype of `List<?>`, you can use `printList()` to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

> *Note: The `Arrays.asList()` method is used in examples throughout this section. This static factory method converts the specified array and returns a fixed-size list.*

It's important to note that `List<Object>` and `List<?>` are not the same. You can insert an `Object`, or any subtype of `Object`, into a `List<Object>`. But you can only insert `null` into a `List<?>`. The Guidelines for Wildcard Use paragraph at the end of this section has more information on how to determine what kind of wildcard, if any, should be used in a given situation.

Lower Bounded Wildcards

The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the `extends` keyword. In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

A lower bounded wildcard is expressed using the wildcard character ('`?`'), following by the `super` keyword, followed by its lower bound: `<? super A>`.

> *Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.*

Say you want to write a method that puts `Integer` objects into a list. To maximize flexibility, you would like the method to work on `List<Integer>`, `List<Number>`, and `List<Object>` — anything that can hold `Integer` values.

To write the method that works on lists of `Integer` and the supertypes of `Integer`, such as `Integer`, `Number`, and `Object`, you would specify `List<? super Integer>`. The term `List<Integer>` is more restrictive than `List<? super Integer>` because the former matches a list of type `Integer` only, whereas the latter matches a list of any type that is a supertype of `Integer`.

The following code adds the numbers 1 through 10 to the end of a list:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

The Guidelines for Wildcard Use paragraph at the end of this section provides guidance on when to use upper bounded wildcards and when to use lower bounded wildcards.

Wildcards and Subtyping

As described in previous sections, generic classes or interfaces are not related merely because there is a relationship between their types. However, you can use wildcards to create a relationship between generic classes or interfaces.

Given the following two regular (non-generic) classes:

```
class A { /* ... */ }
class B extends A { /* ... */ }
```

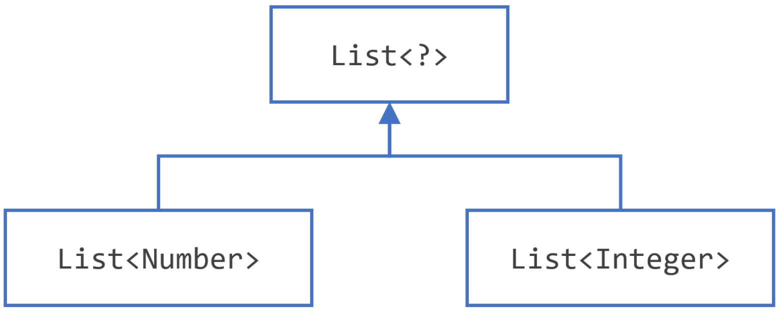
It would be reasonable to write the following code:

```
B b = new B();
A a = b;
```

This example shows that inheritance of regular classes follows this rule of subtyping: class `B` is a subtype of class `A` if `B` extends `A`. This rule does not apply to generic types:

```
List<B> lb = new ArrayList<>();
List<A> la = lb;   // compile-time error
```

Given that `Integer` is a subtype of `Number`, what is the relationship between `List<Integer>` and `List<Number>`?



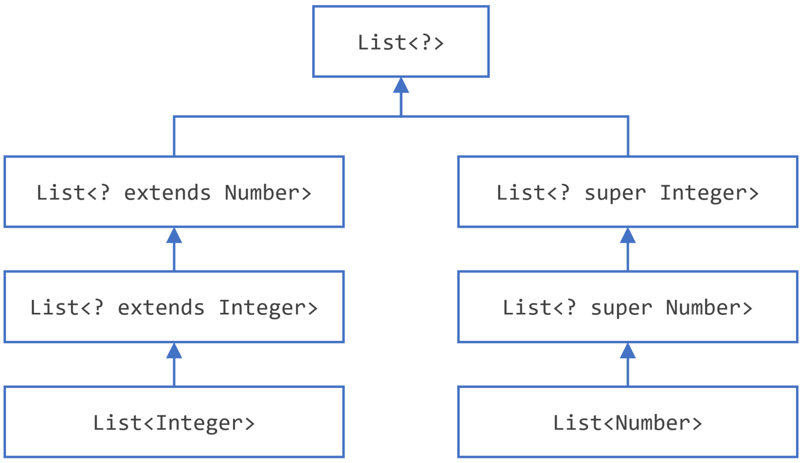
The common parent parameterized lists.

Although `Integer` is a subtype of `Number`, `List<Integer>` is not a subtype of `List<Number>` and, in fact, these two types are not related. The common parent of `List<Number>` and `List<Integer>` is `List<?>`.

In order to create a relationship between these classes so that the code can access `Number`'s methods through `List<Integer>`'s elements, use an upper bounded wildcard:

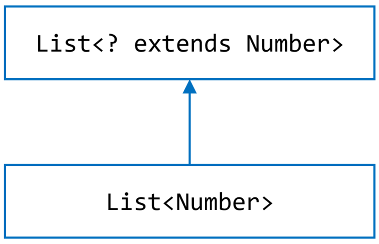
```
List<? extends Integer> intList = new ArrayList<>();
// This is OK because List<? extends Integer> is a subtype of List<? extends Number>
List<? extends Number>  numList = intList;
```

Because `Integer` is a subtype of `Number`, and `numList` is a list of `Number` objects, a relationship now exists between `intList` (a list of `Integer` objects) and `numList`. The following diagram shows the relationships between several `List` classes declared with both upper and lower bounded wildcards.



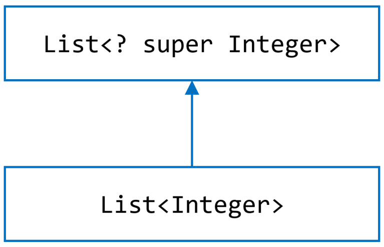
A hierarchy of several generic List class declarations.

Following the same rules, a `List<? extends Number>` can be extended by a list of any type that is an extension of `Number`, including `Number` itself, as shown on the following diagram.



A list of Number extends a list of ? extends Number.

And the same goes for the relationship between `List<? super Integer>` and `List<Integer>`.



A list of Integer extends a list of ? super Integer.

The Guidelines for Wildcard Use paragraph at the end of this section has more information about the ramifications of using upper and lower bounded wildcards.

Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>` but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as wildcard capture.

For the most part, you do not need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The `WildcardError` example produces a capture error when compiled:

```java
import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

In this example, the compiler processes the `i` input parameter as being of type `Object`. When the `foo` method invokes `List.set(int, E)`, the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The `WildcardError` example generates the following error when compiled by Oracle's JDK 7 `javac` implementation:

```
WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a private helper method which captures the wildcard. In this case, you can work around the problem by creating the private helper method, `fooHelper()`, as shown in `WildcardFixed`:

```java
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }


    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

Thanks to the helper method, the compiler uses inference to determine that `T` is `CAP#1`, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named `originalMethodNameHelper()`.

Now consider a more complex example, `WildcardErrorBad`:

```java
import java.util.List;

public class WildcardErrorBad {

    void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {
      Number temp = l1.get(0);
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
                            // got a CAP#2 extends Number;
                            // same bound, but different types
      l2.set(0, temp);        // expected a CAP#1 extends Number,
                            // got a Number
    }
}
```

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the `swapFirst()` method:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double>  ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

While `List<Integer>` and `List<Double>` both fulfill the criteria of `List<? extends Number>`, it is clearly incorrect to take an item from a list of `Integer` values and attempt to place it into a list of `Double` values.

Compiling the code with Oracle's JDK `javac` compiler produces the following error:

```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;
      l2.set(0, temp);      // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;
        i.set(0, i.get(0));
         ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
3 errors
```

There is no helper method to work around the problem, because the code is fundamentally wrong: it is clearly incorrect to take an item from a list of `Integer` values and attempt to place it into a list of `Double` values.

Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard. This page provides some guidelines to follow when designing your code.

For purposes of this discussion, it is helpful to think of variables as providing one of two functions:

- An "In" Variable. An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.
- An "Out" Variable. An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

- An "in" variable is defined with an upper bounded wildcard, using the `extends` keyword.
- An "out" variable is defined with a lower bounded wildcard, using the `super` keyword.
- In the case where the "in" variable can be accessed using methods defined in the `Object` class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by `List<? extends ...>` can be informally thought of as read-only, but that is not a strict guarantee. Suppose you have the following two classes:

```java
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // ...
}

class EvenNumber extends NaturalNumber {

    public EvenNumber(int i) { super(i); }
    // ...
}
```

Consider the following code:

```java
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Because `List<EvenNumber>` is a subtype of `List<? extends NaturalNumber>`, you can assign `le` to `ln`. But you cannot use `ln` to add a natural number to a list of even numbers. The following operations on the list are possible:

- You can add `null`.
- You can invoke `clear()`.

- You can get the iterator and invoke `remove()`.
- You can capture the wildcard and write elements that you have read from the list.

You can see that the list defined by `List<? extends NaturalNumber>` is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

Last update: *September 14, 2021*

Type Inference ← **Wildcards** → Type Erasure

Home > Tutorials > Generics > Wildcards