Wildcards

1. 0 Definition

Wildcards are used to define an unknown type. They provide us further control of type safety when using generics.

1.1 Why are they helpful?

Let's consider the following code.

```
class Animal {
    String name;
    public Animal(String name) {
        this.name = name;
    }
}

class Dog extends Animal {
}

class Cat extends Animal {
}
```

Then, we have a method such as the following:

```
static void printNames(List<Animal> animals) {
    for (Animal animal : animals) {
        System.out.println(animal.name);
    }
}
```

What will happen with the following lines?

```
List<Animal> animals = Arrays.toList(new Dog("Woof"), new
Cat("Meow"));
List<Dog> dogs = Arrays.toList(new Dog("Woof"));
List<Cat> cats = Arrays.toList(new Cat("Meow"));

// will succeed
printNames(animals);
// will error
printNames(dogs);
printNames(cats);
```

You will notice that `printNames(dogs)` will not compile, you will get an error like the following:

> *error: incompatible types: List cannot be converted to List*

This is an issue.

Generics is very strict with the typing. Unlike normal polymorphism...

```
class Animal {
}
class Dog extends Animal {
}

static void foo(Animal animal) {}

// valid function call
foo(new Dog())
```

The same thing cannot be applied with generics.

```
static void foo(List<Animal> animals) {}

List<Dog> dogs = new ArrayList<>();
// invalid function call
foo(dogs);
```

> *error: incompatible types: List cannot be converted to List*

Your first solution may be just to change `dogs` from `List<Dog>` to `List<Animal>`, which is known as polymorphism (treating a child as a parent). This will fix the issue, however, polymorphism is not always applicable. We may need to specifically work with having a list of `Dog` instances, such that we need to use attributes that are not inherited from `Animal`.

Solution?

Wildcards! But before we go into the solution, let's look *just really quickly* how they work. Let's say we have the following two lists.

```
List<String> strings = Arrays.toList("Hello", "World", "!");
List<Integer> integers = Arrays.toList(5, -3, 0);
```

How can I make **one** method that would iterate through each value in the list?

```
static void printElements(List<Object> list) {
    for (Object o : list) {
        System.out.println(o);
    }
}
```

```
printElements(strings);
printElements(integers);
```

This will not work, you will get an error like before:

> *error: incompatible types: List cannot be converted to List*

So instead, we need to do this:

```
static void printElements(List<?> list) {
    for (Object o : list) {
        System.out.println(o);
    }
}
```

We are saying that the list contains an unknown type, but we can guarantee it must be an `Object`.

The two function calls on `strings` and `integers` will now work.

```
List<String> strings = Arrays.toList("Hello", "World", "!");
printElements(strings);
> Hello
  World
  !

List<Integer> integers = Arrays.toList(5, -3, 0);
printElements(integers);
> 5
  -3
  0
```

1.2 But is `?` by itself really helpful?

In short, no. This literally just means you are safe to assume it is an `Object`. There is very little you can do with that information. You can use the instance methods of an `Object` such as `hashCode()`, but that's about it. You wouldn't be able to do anything useful in *most* cases.

So let's go back to our original example.

```
static void printNames(List<Animal> animals) {
    for (Animal animal : animals) {
        System.out.println(animal.name);
    }
}
```

We want to be able to call this using a list of `Dog` or list of `Cat`. Essentially, anything that inherits from `Animal`.

Solution?

We use a wildcard, and say the unknown type *must* extend from `Animal`.

```
static void printNames(List<? extends Animal> animals) {
    for (Animal animal : animals) {
        System.out.println(animal.name);
    }
}
```

Perfecto! This allows the compiler to *type check* that the list passed in has a parameter type that inherits from `Animal`.

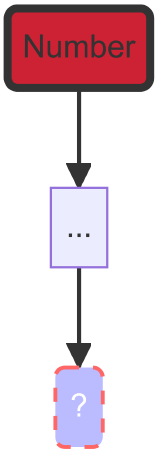
Now let's look at the output.

```
// works for Animal
List<Animal> animals = Arrays.toList(new Dog("Woof"), new
Cat("Meow"));
printNames(animals)
> Woof
  Meow

// also works for anything that inherits from Animal
List<Dog> dogs = Arrays.toList(new Dog("Woof"));
printNames(dogs)
> Woof

List<Cat> cats = Arrays.toList(new Cat("Meow"));
printNames(cats)
> Meow
```

2.0 Upper-bound

```
<? extends SomeClass>
```

We call this an *upper-bound*, since `?` is bounded to be either an instance of `SomeClass` or a child of `SomeClass`.

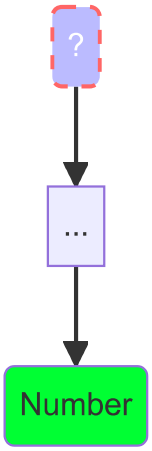


In this instance, we know that the parameter must either be a `Number` instance or a child (inherit) from `Number`.

2.1 Lower-bound

```
<? super SomeClass>
```

We call this a *lower-bound*, since `?` is bounded to be either a `SomeClass` or a parent of `SomeClass`.



In this instance, we know that the parameter must either be a `Number` instance or a parent (superclass) of `Number`.

2.2 In which scenario do you use which?

We can use the following table to help us:

SYNTAX	BOUND	USE

SYNTAX	BOUND	USE
`<? extends SomeClass>`	Lower	Gets
`<? super SomeClass>`	Upper	Puts

2.2.1 Reading

Suppose we have the following:

```
static double sum(Collection<Number> numbers) {
    double result = 0.0;
    for (Number num : numbers) {
        result += num.doubleValue();
    }
    return result;
}
```

We could only pass in `List<Number>` to the function. If we attempted to pass in a `List<Integer>` into the method, the following error will be raised:

> ```
> sum(java.util.Collection<java.lang.Number>) in
> GenericsWildcards cannot be applied to
> (java.util.List<java.lang.Integer>)
> ```

Solution?

We use the `extends` keyword. This is when we want to **read** a collection i.e. **get** values. Let's do an example.

```
// a method that sums up a list of numbers
// since all numbers inherit from Number, we can sum any list of
numbers
static double sum(List<? extends Number> numbers) {
    double sum = 0.0;
    for (Number number : numbers) {
         // doubleValue is an instance method of Number
         // check the javadocs!
         // Q: Why we use doubleValue?
         // A: Because remember that all summations will implictly
cast to double (if needed).
        sum += number.doubleValue();
    }
    return sum;
}
```

We make use of the `extends` keyword as we know that we must pass in a list of numbers, whether it be `Integer`, `Double`, etc. This allows us to do a sum with any list of numbers (as all numbers inherit from `Number`). Hence, this signifies the idea that `extends` is used for **reading** a collection. We often say using `extends` allows us to **get** values.

```
List<Integer> integers = Arrays.toList(3, -5, 20);
sum(integers);
> 18.0

List<Double> doubles = Arrays.toList(3.0, -5.6, 74.3);
sum(doubles);
> 71.7
```

2.2.2 Writing

Suppose we have the following:

```
class Animal {
}
class Cat extends Animal {
}

static void addCat(List<Cat> cats, Cat cat) {
    list.add();
}
```

This method will only take in a `List<Cat>`.

If we pass in a `List<Animal>` or `List<Object>` it will not compile.

```
addCat(new ArrayList<Animal>(), new Cat());
```

> *addCat(java.util.Collection<java.lang.Cat>) in GenericsWildcards cannot be applied to (java.util.List<java.lang.Animal>)*

It may sound confusing, why would we want to pass in a `List<Animal>` or `List<Object>` in the first place?

Remember that `Animal` and `Object` are superclasses of `Cat`, meaning these lists can store these objects. However, the method doesn't allow it, as the collection **must** be only a list that stores the type `Cat`.

Solution?

We use `super` when we want to **write** to a collection i.e. **put** values. Let's do an example.

```
static void addCat(List<? super Cat> cats, Cat cat) {
    list.add(cat);
}
```

Since we do `<? super Cat>`, we are

- The list can store the type `Cat` or a parent of `Cat`, so we could pass in `List<Animal>` and also `List<Object>`.
- We are making it so you can only add `Cat` or child instances of `Cat` into the list.

So because of these rules, let's look at what's good and what's not good.

```
List<Cat> cats = new ArrayList<>();
List<Animal> animals = new ArrayList<>();
List<Object> objects = new ArrayList<>();;

// The 3 function calls below are good
addCat(cats, new Cat());
addCat(animals, new Cat());
addCat(objects, new Cat());

class Ragdoll extends Cat {
}

// The 3 function calls below are good
addCat(cats, new Ragdoll());
addCat(objects, new Ragdoll())
addCat(objects, new Ragdoll());

List<Ragdoll> ragdolls = new ArrayList<>();
```

```
// The function calls below are NOT good, error
addCat(ragdolls, new Cat());
addCat(ragdolls, new Cat());
```

> *error: incompatible types: List cannot be converted to List<? super Cat>*
> *foo(ragdolls, new Cat());*

You can see that there's a strict upper-bound. The list passed in **must** not be a child of `Cat`. This allows it to be safe to write anything into the list that's a `Cat`, or any parent of `Cat`. Do you see the pattern? We often say `super` allows us to **put** values.

3.0 Summary

- Generics must be the exact type. See above examples where passing in `List<Dog>` into a function that accepts `List<Animal>` does not compile.
- We use the `extends` keyword i.e. `<? extends SomeClass>` to do reading. We can assume it must be `SomeClass` or a child of `SomeClass`, allowing us to **get** values.

- We use the `super` keyword i.e. `<? super SomeClass>` to do writing. We can assume it must be `SomeClass` or a parent of `SomeClass`, allowing us to **write** values.