Generics : List<? extends Animal> is same as List<Animal>?

Hi,

I am just trying to understand the `extends` keyword in Java Generics.

`List<? extends Animal>` means we can stuff any object in the `List` which **IS A** `Animal`

then won't the following also mean the same thing:

```
List<Animal>
```

Can someone help me know the difference between the above two? To me `extends` just sound redundant here.

Thanks!

java | generics | extends

asked **Apr 4 at 18:01**

peakit
610 4 14
100% accept rate

3 Answers

`List<Dog>` is a subtype of `List<? extends Animal>` , but not a subtype of `List<Animal>` .

Why is `List<Dog>` not a subtype of `List<Animal>` ? Consider the following example:

```
void mySub(List<Animal> myList) {
    myList.add(new Cat());
}
```

If you were allowed to pass a `List<Dog>` to this function, you would get a run-time error.

EDIT: Now, if we use `List<? extends Animal>` instead, the following will happen:

```
void mySub(List<? extends Animal> myList) {
    myList.add(new Cat());     // compile error here
    Animal a = myList.get(0);  // works fine
}
```

You *could* pass a `List<Dog>` to this function, but the compiler realizes that adding something to the list could get you into trouble. If you use `super` instead of `extends` (allowing you to pass a `List<LifeForm>`), it's the other way around.

```
void mySub(List<? super Animal> myList) {
    myList.add(new Cat());     // works fine
    Animal a = myList.get(0);  // compile error here, since the list entry could b
}
```

The theory behind this is Co- and Contravariance.

edited **Apr 4 at 18:43**

answered **Apr 4 at 18:21**

Heinzi
12.9k 2 13 38

2 +1 For the wikipedia link. – Helper Method Apr 4 at 19:19

It is not. `List<Animal>` says that the value which is assigned to this variable must be of "type" `List<Animal>` . This however doesn't mean that there must only be `Animal` objects, there can be subclasses too.

```
List<Number> l = new ArrayList<Number>();
l.add(4); // autoboxing to Integer
l.add(6.7); // autoboxing to Double
```

You use the `List<? extends Number>` construct if you are interest in an list which got `Number` objects, but the List object itself doesn't need to be of type `List<Number>` but can any other list of subclasses (like `List<Integer>`).

This is sometime use for method arguments to say *"I want a list of `Numbers` , but I don't care if it is just `List<Number>` , it can be a `List<Double>` too"*. This avoid some weird down casts if you have a list of some subclasses, but the method expects a list of the baseclass.

```
publid void doSomethingWith(List<Number> l) {
    ...
}

List<Double> d = new ArrayList<Double>();
doSomethingWith(d); // not working
```

This is not working as you expecting `List<Number>` , not a `List<Double>` . But if you wrote `List<? extends Number>` you can pass `List<Double>` objects even as they aren't `List<Number>` objects.

```
publid void doSomethingWith(List<? extends Number> l) {
    ...
}

List<Double> d = new ArrayList<Double>();
doSomethingWith(d); // works
```

Note: This whole stuff is unrelated to inheritance of the objects in the list itself. You still can add `Double` and `Integer` objects in a `List<Number>` list, with or without `? extends` stuff.

edited **Apr 4 at 19:50**
AlBlue
3,523 3 17

answered **Apr 4 at 18:30**
Progman
838 1 2 9

hmmm.. getting it. Nice explanation – peakit Apr 4 at 18:38

I see you've already accepted an answer, but I'd just like to add my take on it, as I think I can be of some help here.

The difference between `List<Animal>` and `List<? extends Animal>` is this:

With `List<Animal>` , you know what you have is definitely a list of animals. It's not necessary for all of them to actually be exactly 'Animal's - they could also be derived types. For example, if you have a List of Animals, it makes sense that a couple could be Goats, and some of them Cats, etc - right?

For example this is totally valid:

```
List<Animal> aL= new List<Animal>();
aL.add(new Goat());
aL.add(new Cat());
Animal a = aL.peek();
a.walk();//assuming walk is a method within Animal
```

Just a sidenote - the following would *not* be valid:

```
aL.peek().meow();//we can't do this, as it's not guaranteed that aL.peek() will be
```

```
((Cat)aL.peek()).meow();//will generate a runtime error if aL.peek() is not a Cat
```

With `List<? extends Animal>` , you're making a statement about the *type of list* you're dealing with.

For example:

```
List<? extends Animal> L;
```

This is actually *not* a declaration of the type of object L can *hold*. **It's a statement about what kinds of lists L can reference.**

For example, at this point,

```
L = aL;//remember aL is a List of Animals
```

would be something we could do.

But even after that assignment, all the compiler knows about L is that it is a *List of [either Animal or a subtype of Animal]s*

So now the following is not valid:

```
L.add(new Animal());//throws a compiletime error
```

Because for all we know, L could be referencing a list of Goats - to which we absolutely cannot add an Animal.

Why not? Well, let's see:

```
List<Goat> gL = new List<Goat>();//fine
gL.add(new Goat());//fine
gL.add(new Animal());//compiletime error
```

The reason the above doesn't work is we are attempting to cast an Animal as a Goat. That doesn't work, because what if after doing that we tried to make that Animal do a 'headbutt', like a goat would? We don't necessariliy know that the Animal can do that.

edited Apr 4 at 19:37

answered **Apr 4 at 19:24**

Cam - incrediman

2,711 2 21

+1 for "This is actually not a declaration of the type of object L can hold. It's a statement about what kinds of lists L can reference." – Heinzi Apr 4 at 20:01