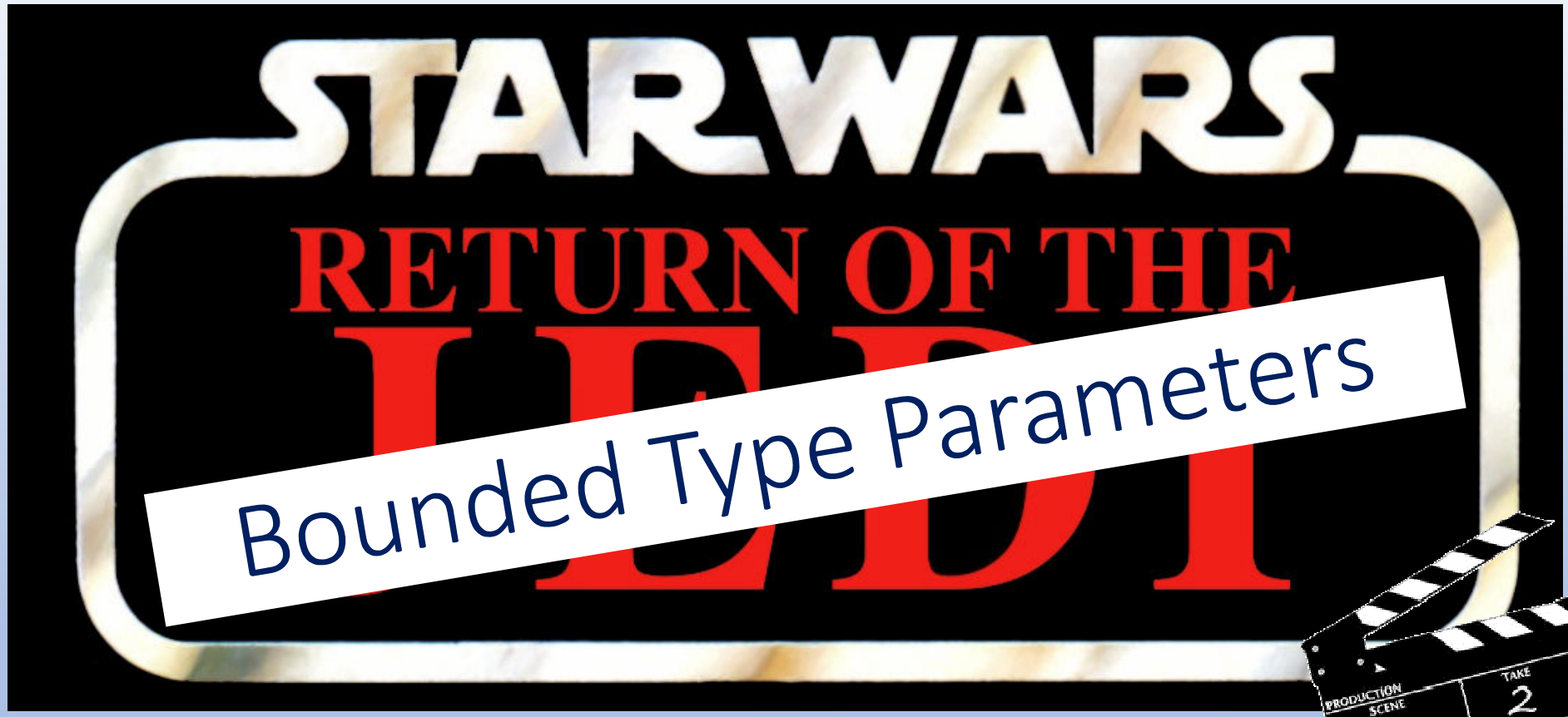

Bounded Type Parameters

So you think you got it?

Wait a minute…

May everything we did in our Bounded Types example…

…be done w/ only polymorphism???

<U extends MyClass> vs. MyClass as parameter


```
public class JustTesting{
    public static void main(String[] args){
        JustTesting jt = new JustTesting();
        Integer datum1 = new Integer(42);
        Double datum2 = new Double(99.9);
        jt.display1(datum1);
        jt.display1(datum2);
        jt.display2(datum1);
        jt.display2(datum2);
    }

    public <U extends Number> void display1(U p){System.out.println(p);}

    public void display2(Number p){System.out.println(p);}
}
```

Alright but if we use the data type several times in the method, then Generics beat using the superclass…

…right?

What if we use many times the data type?

```
public class JustTesting{
   public static void main(String[] args){
      JustTesting jt = new JustTesting();
      Integer datum1 = new Integer(42);
      Double datum2 = new Double(99.9);

      jt.display1(datum1, datum2);
      jt.display2(datum1, datum2);
   }

   public <U extends Number> void display1(U p1, U p2){
      System.out.println(p1 + " " + p2);
   }

   public void display2(Number p1, Number p2){
      System.out.println(p1 + " " + p2);
   }
}
```

Both work!
U is inferred to be just Number

We want to force the same subtype to be used for p1 and p2

Here, clearly, we could pass an Integer and a Double as p1 and p2

Basic examples fail…
so what would work then???

However…

```
// we need a class and subclass
class Bar extends Foo { }

// two methods from anohter class, as before
public <T extends Foo> void doSomething1(List<T> foos) {}
public void doSomething2(List<Foo> foo) {}


// then we try to use each method as follows;
List<Bar> list = new ArrayList<Bar>();

doSomething1(list);

doSomething2(list);
```

So this makes a difference!

valid for 1st method
type parameter T inferred as Bar

2nd method fails because a List<Foo> is not a super type of List<Bar>,
although Foo is super type of Bar.

stack overflow https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

Basic examples fail…
so what would work then???

However…

```
// we need a class and subclass
class Bar extends Foo { }

// two methods from anohter class, as before
public <T extends Foo> void doSomething1(List<T> foos) {}
public void doSomething2(List<Foo> foo) {}


// then we try to use each method as follows;
List<Bar> list = new ArrayList<Bar>();

doSomething1(list);

doSomething2(list);
```

…We did not replace a superclass by a type parameter extending it…

…So we answered a *different question…*

What would be an actual solution here?

List<? extends Foo>

https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

Let's go back to having more than one use for the type variable…

This time we have T in a List<…>

```
public static <T> void addToList(List<T> list, T element) {list.add(element);}

List<Integer> list = new ArrayList<>();

addToList(list, 7);
```

✓ This was expected though…

stack overflow https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

Works also with unrelated types

```
public static <T> void addToList(List<T> list, T element) {list.add(element);}

List<Integer> list = new ArrayList<>();

addToList(list, "a");
```

```
error: method addToList in class JustTesting cannot be
applied to given types;
      addToList(list, "a");
      ^
  required: List<T>,T
  found: List<Integer>,String
  reason: inference variable T has incompatible bounds
    equality constraints: Integer
    lower bounds: String
  where T is a type-variable:
    T extends Object declared in method
<T>addToList(List<T>,T)
1 error
```

This was expected though…

https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

However, now Integer vs. Double also matters!

```
public static <T> void addToList(List<T> list, T element) {list.add(element);}

List<Integer> list = new ArrayList<>();

addToList(list, 0.7);
```

```
error: method addToList in class JustTesting cannot be
applied to given types;
      addToList(list, 0.7);
      ^
  required: List<T>,T
  found: List<Integer>,double
  reason: inference variable T has incompatible bounds
    equality constraints: Integer
    lower bounds: Double
  where T is a type-variable:
    T extends Object declared in method
<T>addToList(List<T>,T)
1 error
```

! Type inference unable to find Number as common denominator here…

…makes sense as List<Number> is not superclass of List<Integer>

stack overflow https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

When using T multiple times in a generic method…

…It is also possible to try to match a parameter (or more) with the return value

Another example
parameter & return type must be the same

```
public static <T> T nullCheck(T value, T defValue) {
    return value != null ? value : defValue;
}

Integer iN = null;
Integer i = nullCheck(iN, 7);
System.out.println(i); // "7"
```

As expected…

https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

Another example
parameter & return type must be the same

```
public static <T> T nullCheck(T value, T defValue) {
    return value != null ? value : defValue;
}

Integer iN = null;
Integer i = nullCheck(iN, 7);
System.out.println(i); // "7"

Double dN = null;
Double d = nullCheck(dN, 0.7);
System.out.println(d); // "0.7"
```

As expected…

https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

Another example
parameter & return type must be the same

```
public static <T> T nullCheck(T value, T defValue) {
    return value != null ? value : defValue;
}

Integer iN = null;
Integer i = nullCheck(iN, 7);
System.out.println(i); // "7"

Double dN = null;
Double d = nullCheck(dN, 0.7);
System.out.println(d); // "0.7"

Number n = nullCheck(i, d); // T = superclass of Integer and Double
System.out.println(n); // "7"
```

Type inference finds Number as common denominator here…

https://stackoverflow.com/questions/21390685/upper-bounded-generics-vs-superclass-as-method-parameters

What if we end up downcasting?

```
public abstract class Animal {
    public abstract void eat();
    public abstract void talk();
}
class Dog extends Animal {
    @Override public void eat() {…}

    @Override public void talk() {…}
}

class Cat extends Animal {
    @Override public void eat() {…}

    @Override public void talk() {…}
}
```

```
public static
<T extends Animal> T replicate(T animal) {
    …
}

public static
Animal replicatePoly(Animal animal) {
    …
}
```

Returns exact same type as parameter

Returns only an Animal, so we need to Downcast it to whatever the parameter was (e.g. Dog or Cat)

https://softwareengineering.stackexchange.com/questions/227918/java-use-polymorphism-or-bounded-type-parameters

So when do we use Generics?

So just use regular polymorphism if you just want to handle specific classes as their base type. Upcasting is a good thing. But if you get in a situation where you need to handle specific classes as their own type, generically, use generics.

Or, really, if you find you have to downcast then use generics.

https://softwareengineering.stackexchange.com/questions/227918/java-use-polymorphism-or-bounded-type-parameters