

Covariance and contravariance of parameterized types

Jean-Luc Delarbre

Definition

- **A generic type is a type with formal type parameters. A parameterized type is an instantiation of a generic type with actual type arguments.**

[Angelika Langer - Java Generics FAQs]

- **Also called parametric polymorphism**

Generic type	Parameterized type
<pre>// E is the (formal) type parameter interface List<E> { public void add(E e); E get(int I); //... }</pre>	<pre>// String is the type argument (or Actual type parameter) List<String></pre>

Raw type = List



Generic purpose

- **Objects handling**

- Collection API
- WeakReference, SoftReference...

- **Type safety**

- Early error detection and avoid cast exception without visible cast

```
List<String> myList = new ArrayList<>();  
myList.add("aString");  
String str = myList.get(0); // Hidden cast to String
```



Generic type creation

```
Class MyGenericType<T, U> { // At least one parameter
    private T t;
    private U u;

    public MyGeneric(T t, U u) {
        this.t = t;
        this.u = u;
    }
    public T getT() {return t;}
    public U getU() {return u;}
    // ...
}
```

```
MyGenericType<String, Double> mgt = new MyGenericType<>("a", 1.0);
```

- Not possible with anonymous class, exceptions or enum. Not usable to define arrays.



Bounded type parameter

- **Restrict the type of a parameter type**

```
class MyType<T extends Number> {  
    T t;  
    int getNearestInt() {  
        return t.intValue(); // Can use "t" as a Number  
    }  
}  
  
MyType<Double> d = new MyType<>();  
MyType<Integer> i = new MyType<>();  
MyType<String> s = new MyType<>(); // Error
```

- **Multiple bounds allowed**

```
<T extends A & B & C & ...>  
ComparableCollectionBox<T, C extends Collection<T> & Comparable<C>> {  
    //...  
}
```



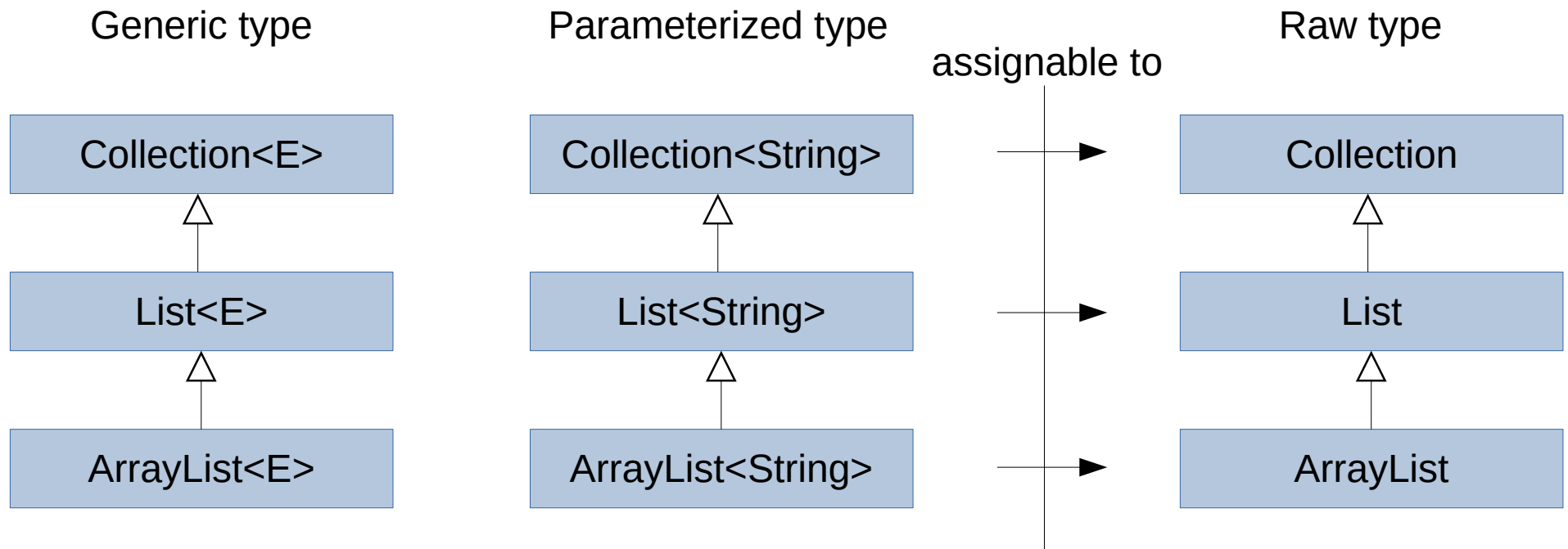
Type erasure

- **Type erasure**
 - Less type information on runtime than compile time
- **Reification**
 - Same type information on runtime than compile time

Reified type	(Generic) Type erasure
<pre>// .Upcast done on compile time Object o = new String(""); // .Type information available on // runtime (o.getClass()) // .Downcast done at runtime String s = (String) o;</pre>	<pre>// .Upcast done on compile time Object o = new ArrayList<String>(); // .Generic type argument information // NOT available on runtime // .Downcast only done at runtime // towards RAW type List l = (List) o; // .Downcast not fully checked List<String> ls = (List<String>) o;</pre>



Inheritance and parameterized type (1/4)

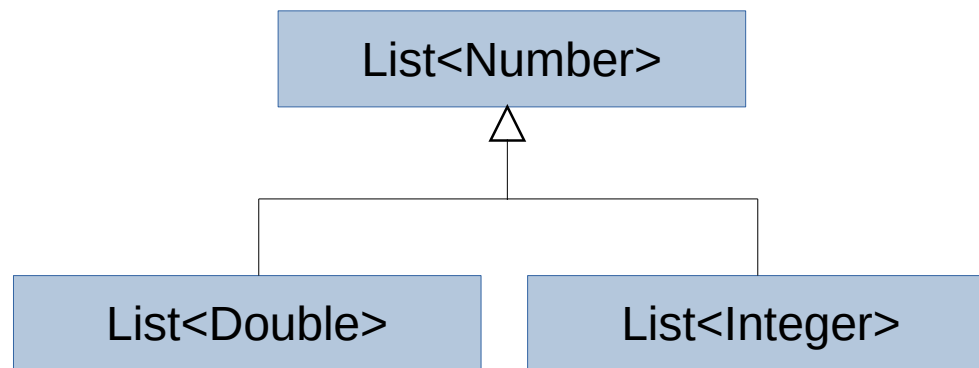
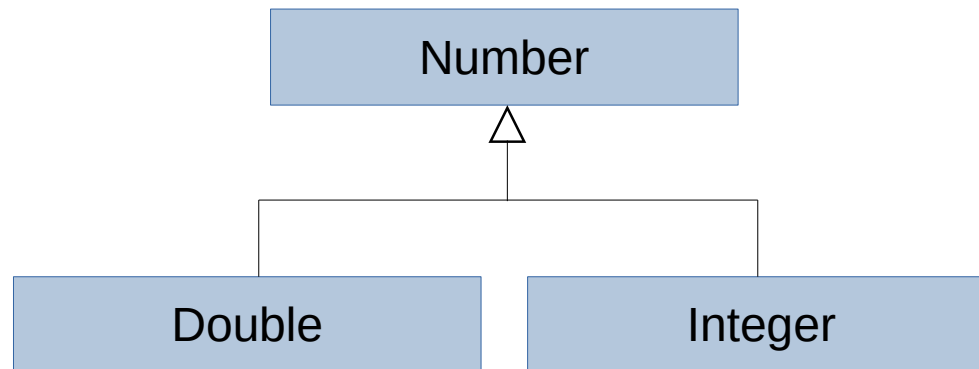


// Safe but not safely reversible

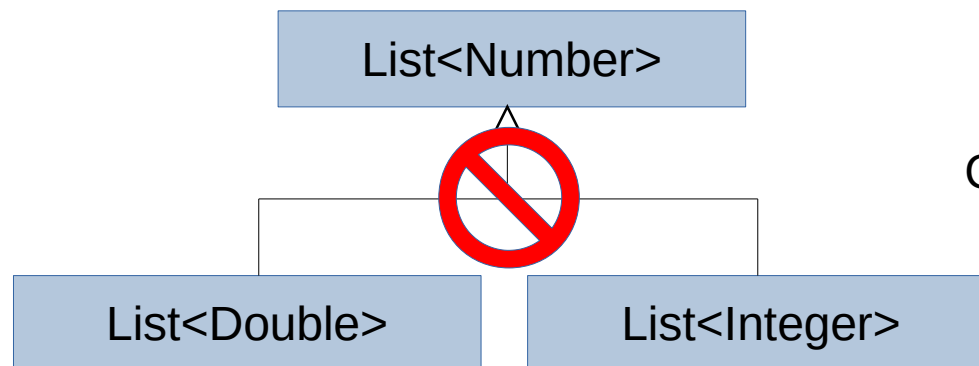
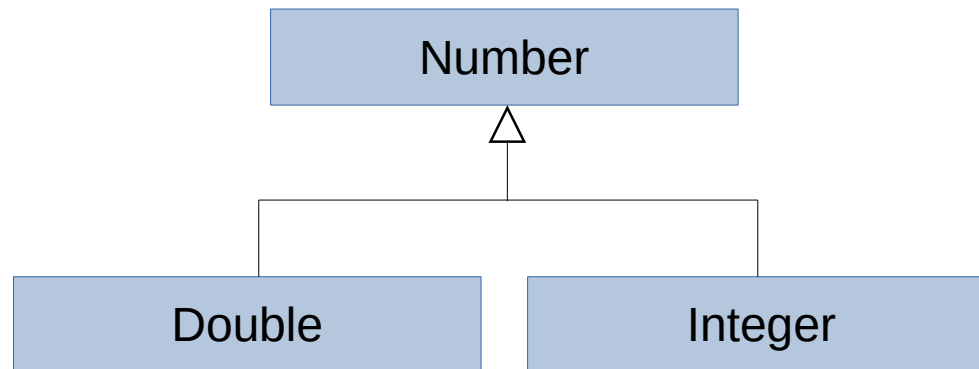
```
List<String> stringList;  
@SuppressWarnings("rawtypes")  
List rawList = stringList;
```



Inheritance and parameterized type (2/4)



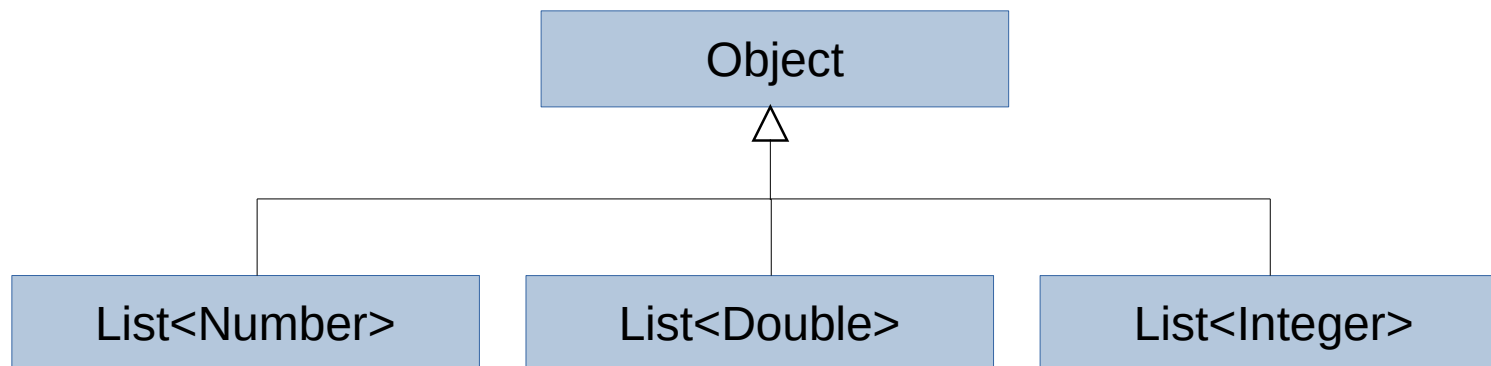
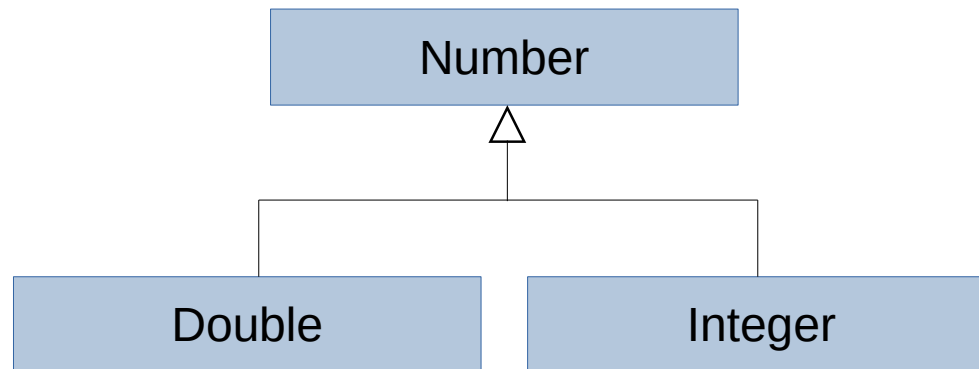
Inheritance and parameterized type (2/4)



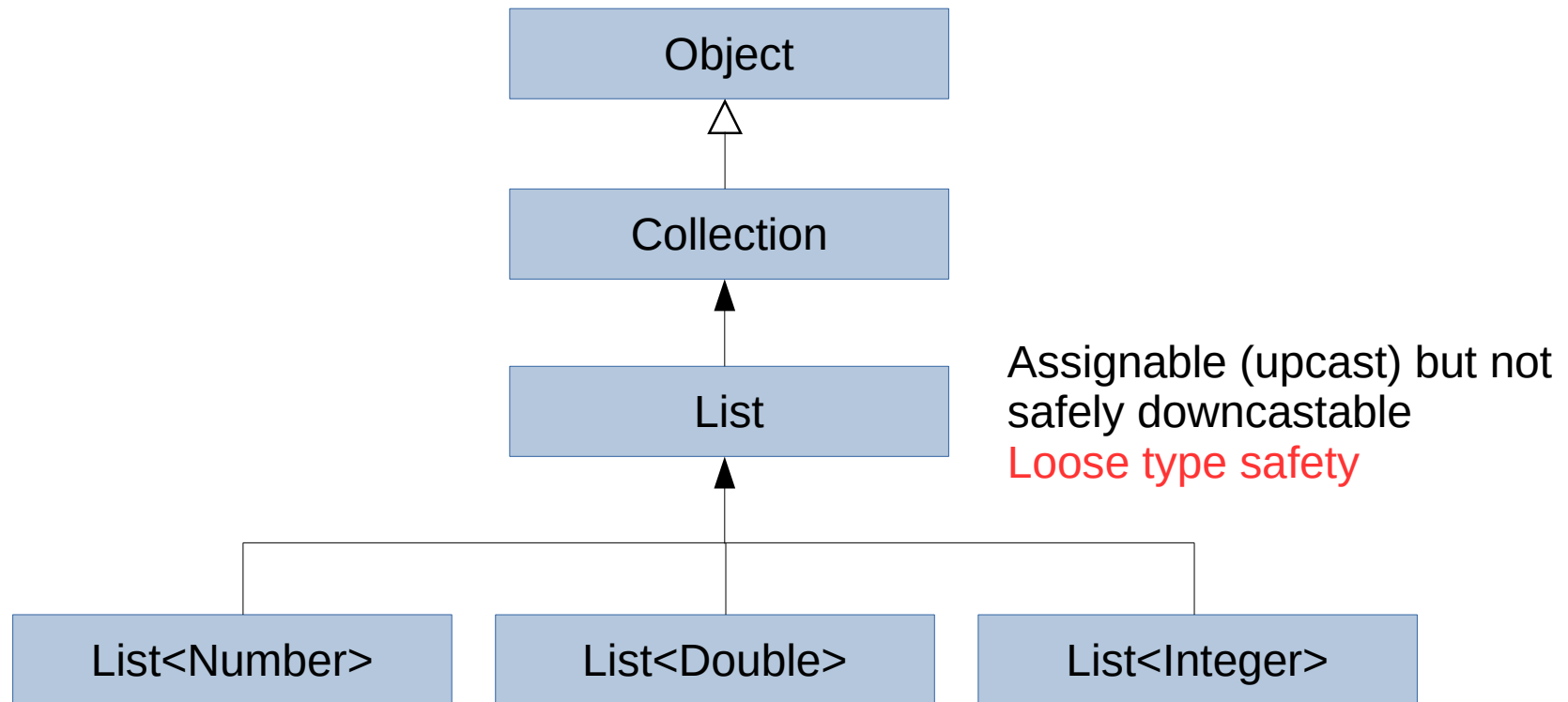
Generics are invariant



Inheritance and parameterized type (3/4)



Inheritance and parameterized type (4/4)



Wildcard

- **“?” is a special type parameter that ensure/control type safety of the use of generic/parameterized type.**
 - Different instantiations of generic type (`List<String>`, `List<Double>`, `List<Number>`) are never compatible. No inheritance relationship.
 - Wildcard “?” allows a certain amount of compatibility between various generic type instantiations.
 - Kind of inheritance with restrictions.
 - Definition of an abstract super type for any parameterized type of a given generic type.

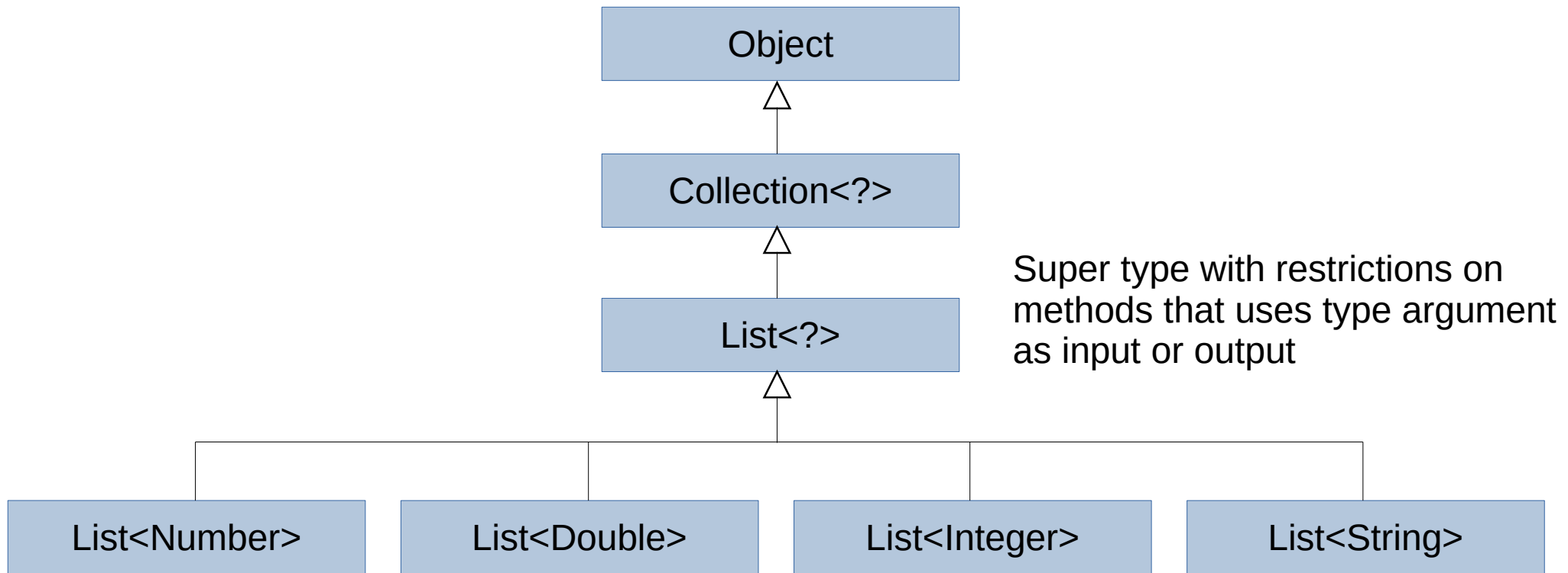


A difficult topic

- **Joshua Bloch criticized wildcard as being too hard to understand and use.**
- **Scala (language on JVM) first uses declaration-site (“concurrent of wildcard”).**
 - Martin Odersky adds wildcard (used-site) only for Java compatibility.



Inheritance and super type with wildcard



```
List<?> list = new ArrayList<Double>();
```

```
list.add(null);
```

```
list.add(5.0); // Error
```

```
Object x = list.get(0); // Only get Object
```

```
// Methods that does not use type argument (Double)  
// Wildcard super types are less capable
```

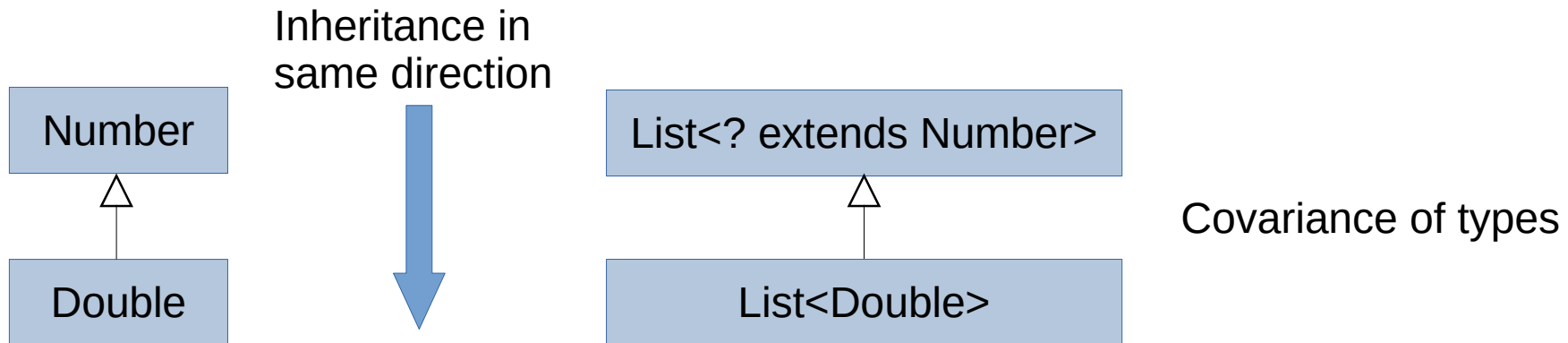
```
list.remove(3);
```

```
list.clear();
```

```
int size = list.size();
```



Wildcard with upper bound



```
List<? Extends Number> list = new ArrayList<Double>(); // Methods that are still usable

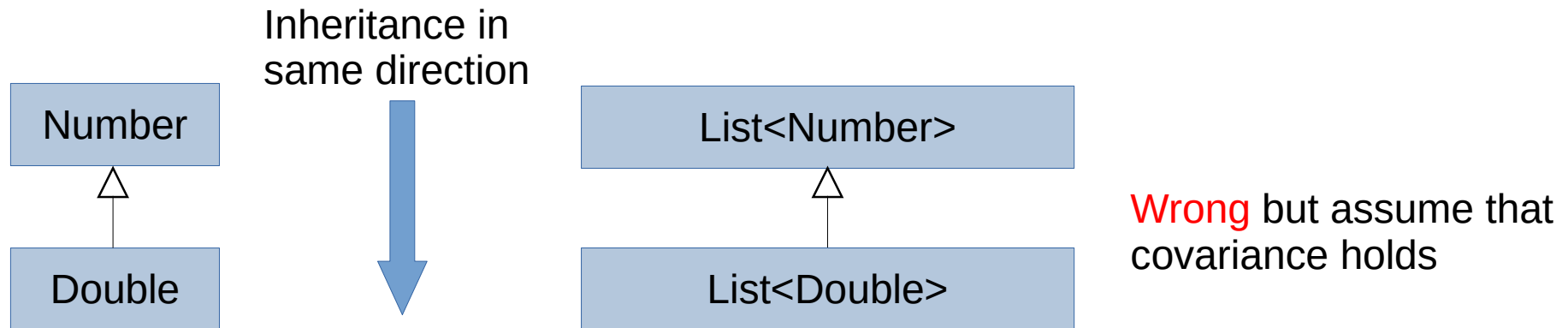
list.add(null);
list.add(5.0); // Error still not usable

Number x = list.get(0); // Known as Number

list.remove(3);
list.clear();
int size = list.size();
```

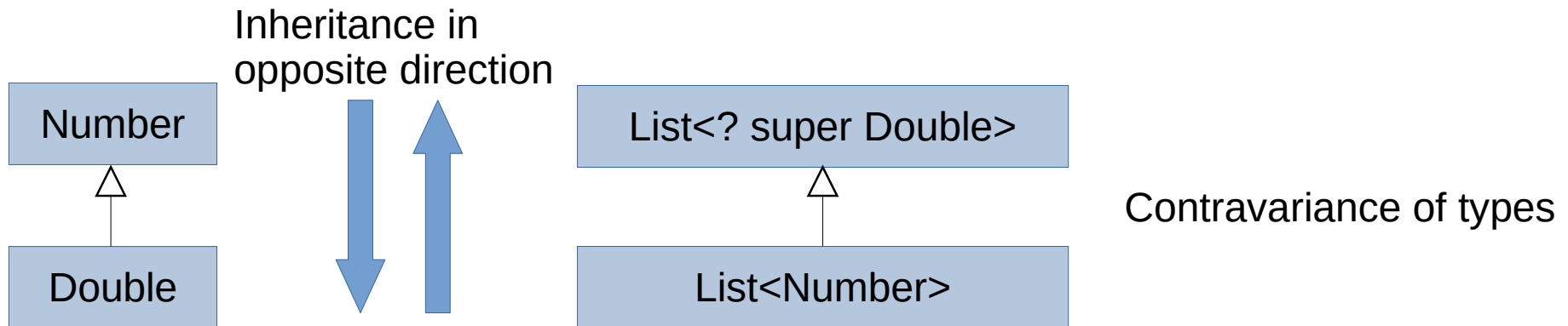


Reason of wildcard limitation (covariant case)



```
List<Double> doubleList = new ArrayList<Double>();  
List<Number> numberList = doubleList; // Forbidden by compiler  
  
// Possible this way but with an unchecked cast (heap pollution)  
@SuppressWarnings("unchecked")  
List<Number> numberList = (List<Number>) (List<?>) doubleList;  
  
numberList.add(Integer.valueOf(3)); // Work  
  
Double x = doubleList.get(0); // Cast error since doubleList contains an Integer
```


Wildcard with lower bound



```
List<Number> nList = new ArrayList<Number>();  
List<? super Double> dList = nList;
```

```
dList.add(5.0); // Only possible to add Double in Number List  
dList.add(3);  // Error do not know the real List type
```

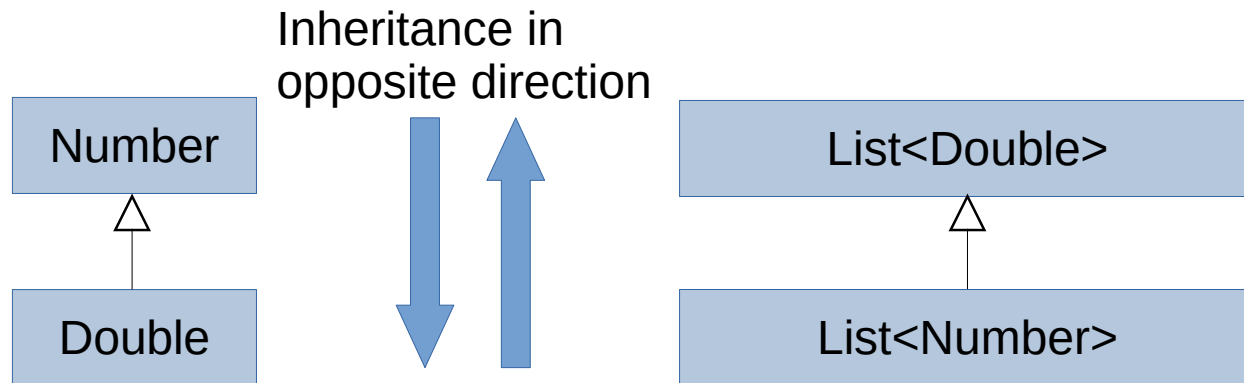
```
Object x = dList.get(0); // Only known as Object  
Double d = dList.get(0); // Not necessarily a Double  
// nList.add(3);
```

```
// Methods that are still usable
```

```
list.remove(3);  
list.clear();  
int size = list.size();
```



Reason of wildcard limitation (contravariant case)



Wrong but assume that contravariance holds

```
List<Number> numberList = new ArrayList<Number>();  
List<Double> doubleList = numberList; // Forbidden by compiler
```

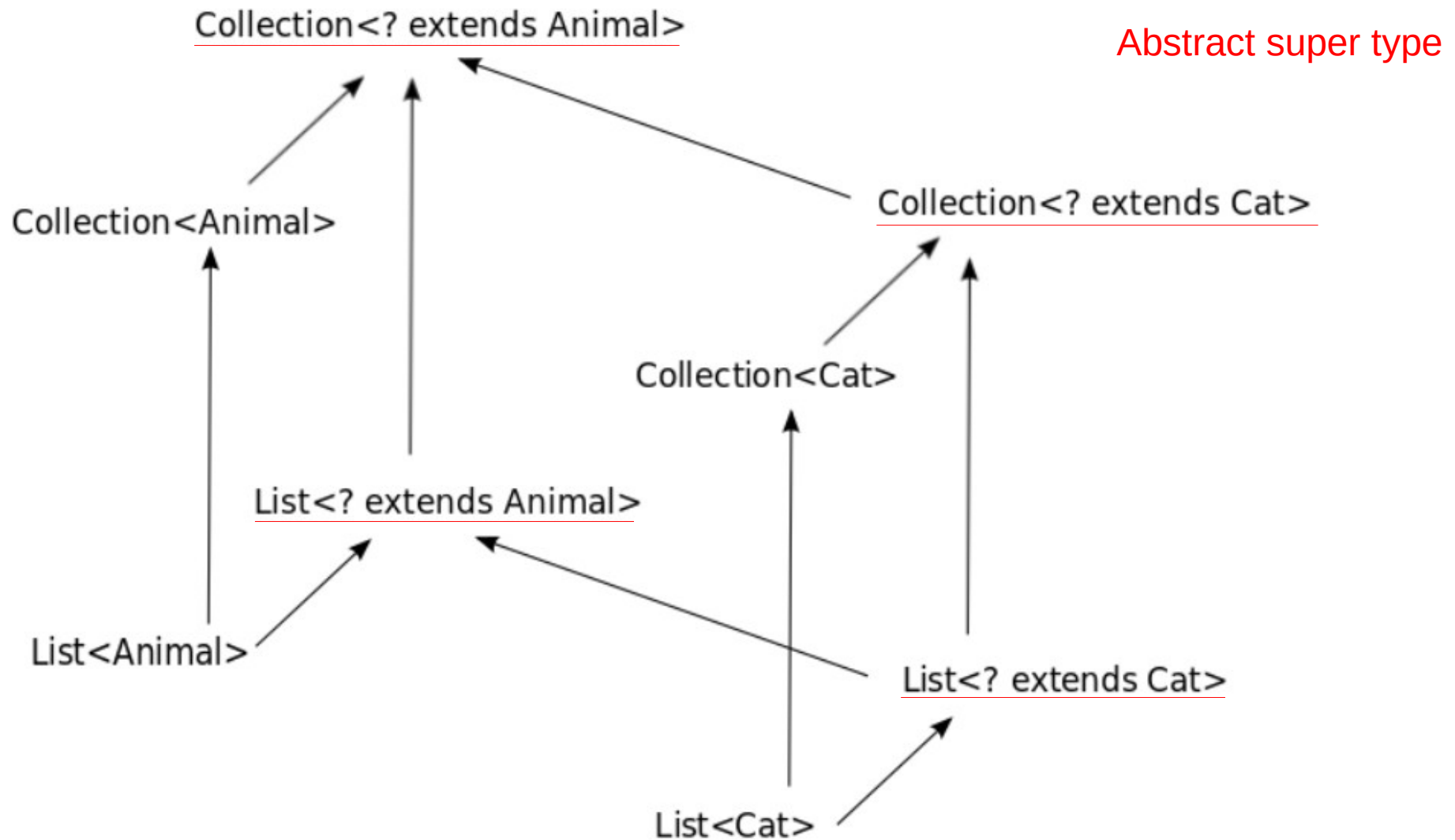
```
// Possible this way but with an unchecked cast (heap pollution)  
@SuppressWarnings("unchecked")  
List<Double> doubleList = (List<Double>) (List<?>) numberList;
```

```
numberList.add(Integer.valueOf(3)); // Normally work as expected  
doubleList.add(Double.valueOf(2.0)); // Normally work as expected
```

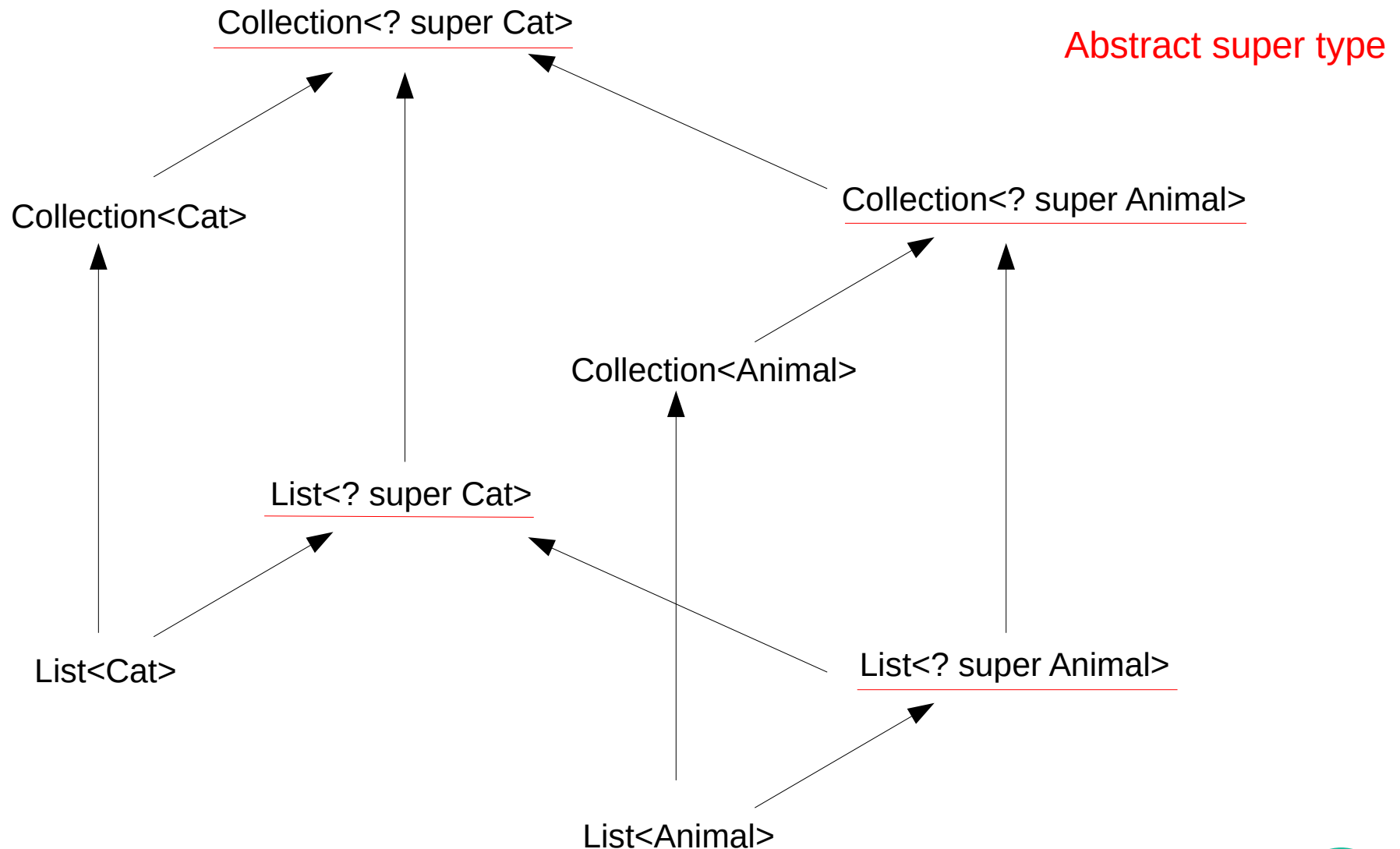
```
Double x = doubleList.get(0); // Cast error since doubleList is a number List that contains an  
// Integer
```



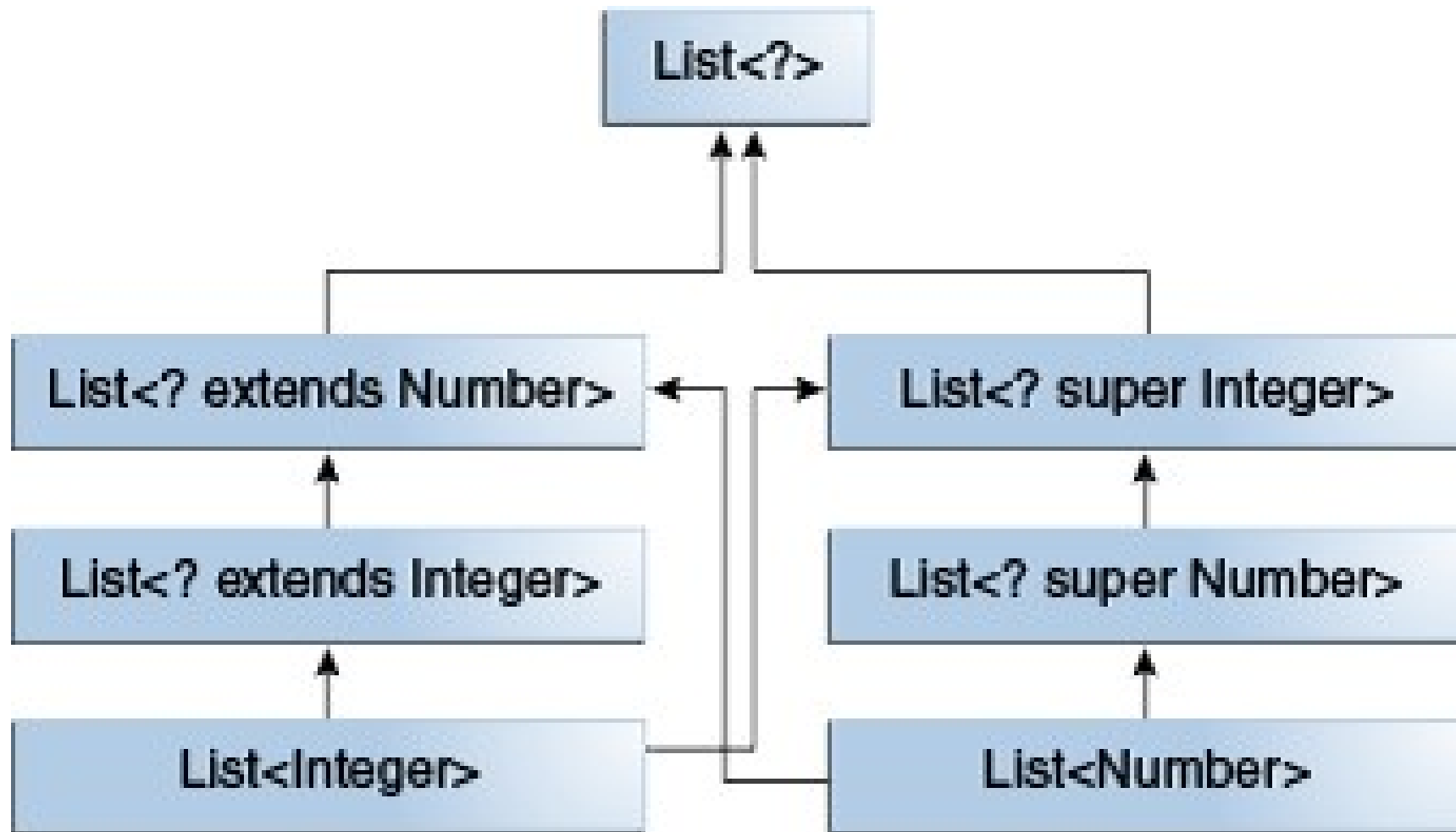
Wildcard subtyping in Java can be visualized as a cube (covariant)



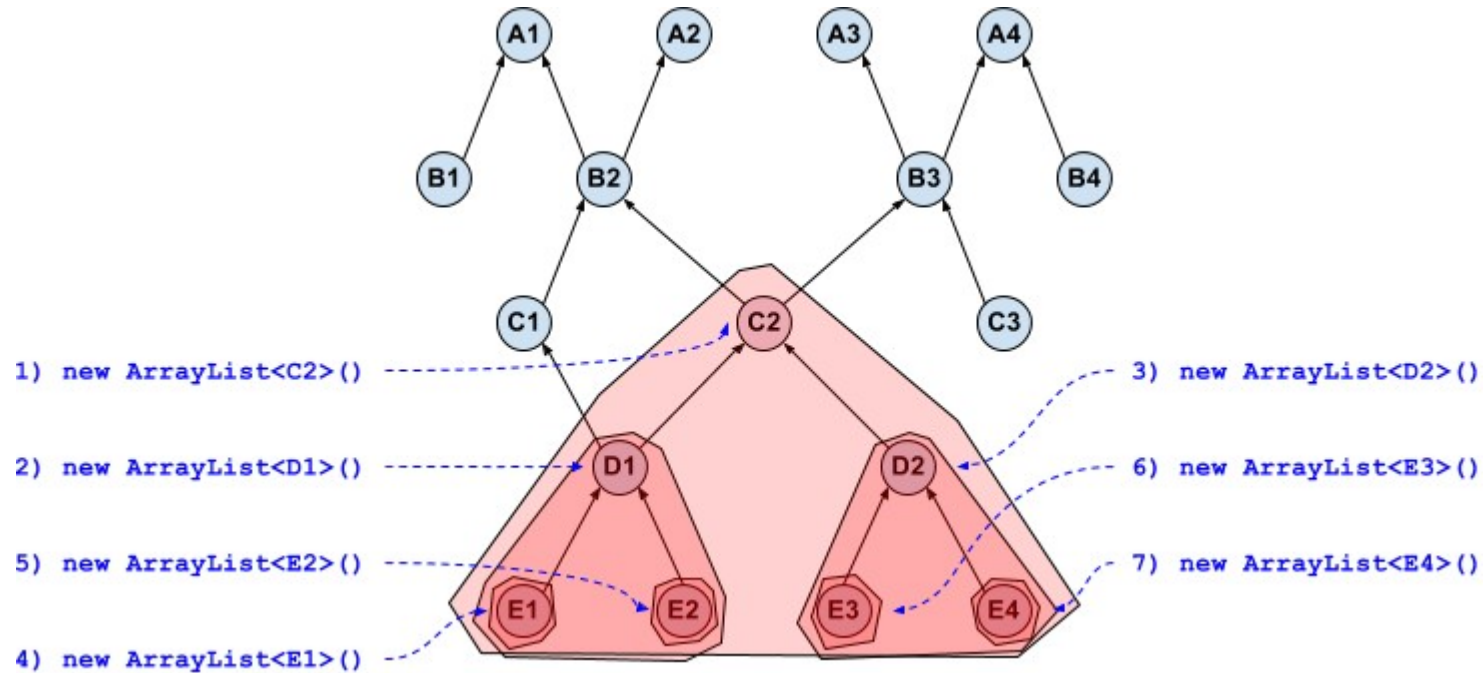
Wildcard subtyping in Java can be visualized as a cube (contravariant)



A hierarchy of several generic List class declarations



Hierarchy for “? extends” super type



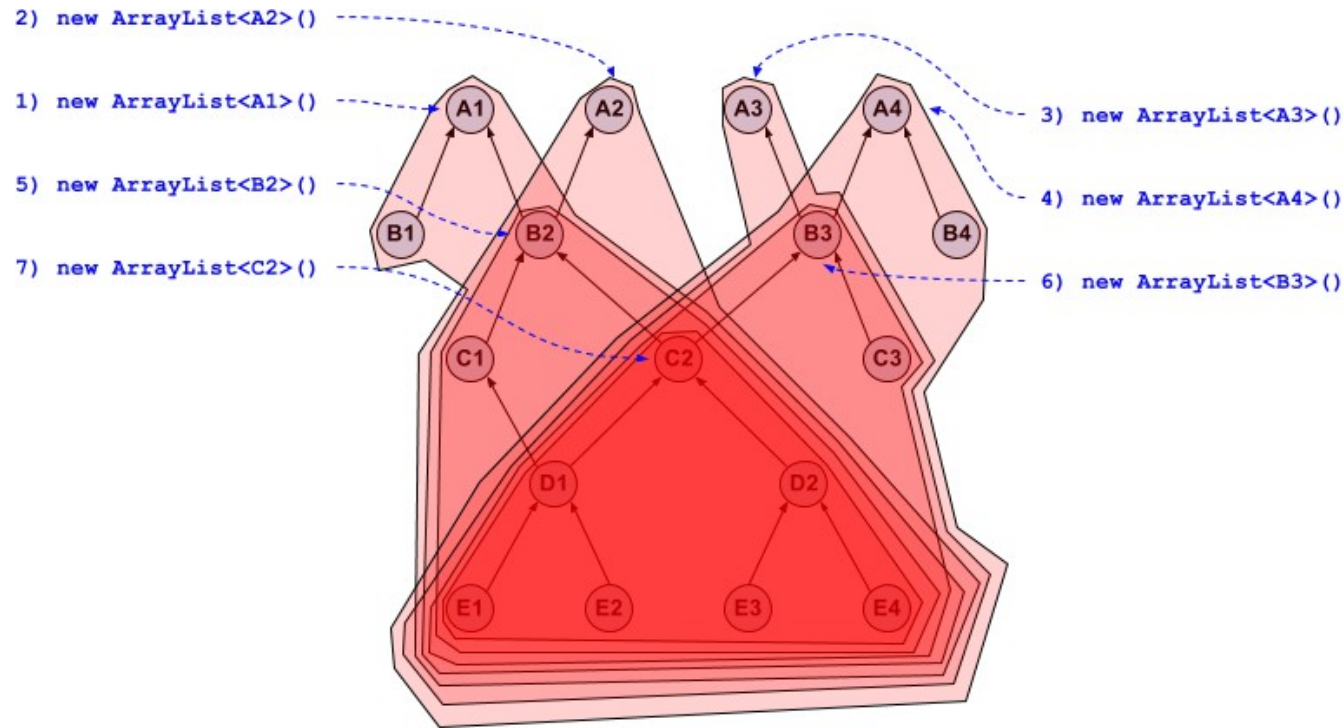
Possible concrete types for `List<? extends C2>`:

1) <code>new ArrayList<C2>()</code> :	can store	C2	D1	D2	E1	E2	E3	E4
2) <code>new ArrayList<D1>()</code> :	can store		D1		E1	E2		
3) <code>new ArrayList<D2>()</code> :	can store			D2			E3	E4
4) <code>new ArrayList<E1>()</code> :	can store				E1			
5) <code>new ArrayList<E2>()</code> :	can store					E2		
6) <code>new ArrayList<E3>()</code> :	can store						E3	
7) <code>new ArrayList<E4>()</code> :	can store							E4

No common parameter type:

- Impossible to write
- Only common super type: C2
 - Could only be read as C2

Hierarchy for “? super” super type



Common parameter types:
C2, D1, D2, E1, E2, E3, E4

- Can write safely all common parameter types
- No common super type
 - Could only be read as Object

Possible concrete types for List<? super C2>:

1)	new ArrayList<A1>():	can store	A1		B1	B2		C1	C2		D1	D2	E1	E2	E3	E4	
2)	new ArrayList<A2>():	can store	A2			B2		C1	C2		D1	D2	E1	E2	E3	E4	
3)	new ArrayList<A3>():	can store		A3			B3		C2	C3	D1	D2	E1	E2	E3	E4	
4)	new ArrayList<A4>():	can store			A4		B3	B4		C2	C3	D1	D2	E1	E2	E3	E4
5)	new ArrayList<B2>():	can store				B2			C1	C2		D1	D2	E1	E2	E3	E4
6)	new ArrayList<B3>():	can store					B3			C2	C3	D1	D2	E1	E2	E3	E4
7)	new ArrayList<C2>():	can store								C2		D1	D2	E1	E2	E3	E4

Usage of abstract super types

```
class Triangle extends Ishape {...}
class Circle extends Ishape {...}

class Canvas {
    List<IShape> shapes;

    void addShape(IShape shape) {
        shapes.add(shape);
    }

    void draw() {
        for (Ishape shape : Shapes) {
            shape.draw();
        }
    }
}
```

- **Writing code that depends on abstraction and is independent of concrete things**



Usage of generic abstract super type (covariant case)

```
interface IBlinkingShape extends IShape {...}

class Canvas {
    List<IShape> shapes;
    //...
    void addAll(List<? Extends IShape> shapes) {
        this.shapes.addAll(shapes);
    }
}

List<IBlinkingShape> blinkingShapes = ...
List<IShape> shapes = ...

canvas.addAll(blinkingShapes);
canvas.addAll(shapes);
```

- **Writing code that depends on abstraction and is independent of concrete things**
 - `List<? extends IShape>` is an abstraction of `IShape` provider/producer
 - `List<IShape>`, `List<IBlinkingShape>` are concrete `IShape` provider/producer

Usage of generic abstract super type (contravariant case)

```
interface IShape extends IDrawable // Image, text, shape...
interface ITransformer<T> {
    void transform(T t) { // Translation, rotation... }
}

class Canvas {
    List<IShape> shapes;
    //...
    void transformShapes(ITransformer<? super IShape> transformer) {
        this.shapes.forEach(transformer::transform);
    }
}

ITransformer<IDrawable> drawableTransformer = ...
ITransformer<IShape> shapeTransformer = ...

canvas.transformShapes(drawableTransformer);
canvas.transformShapes(shapeTransformer);
```

- **Writing code that depends on abstraction and is independent of concrete things**

- `ITransformer<? super IShape>` is an abstraction of `IShape` consumer
- `ITransformer<IShape>`, `ITransformer<IDrawable>` are concrete `IShape` consumer



Usage of wildcards

- **For maximum flexibility, use wildcard types on input parameters that represent producers (output) or consumers (input parameter).**
- **PECS stands for producer-extends, consumer-super.**
- **For both producer and consumer**
 - Use exact type (no wildcard)
- **Do not use bounded wildcard as return types.**



Generic methods

- **Not seen here, but easy to find information in reference**



Another approach to generic types: C#

IEnumerator<T>
T get(); Bool MoveNext();

public interface IEnumerable<out T>

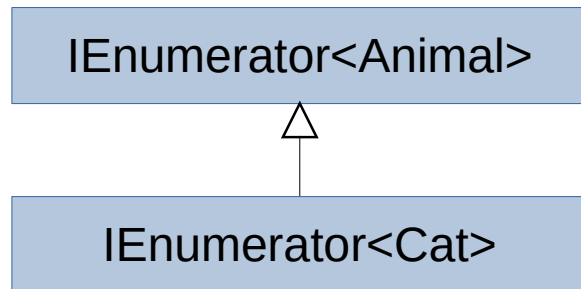
“out” keyword means:

T will only be used as return value



Covariant type

So:



IComparable<T>
int compareTo(T other);

public interface IComparable<in T>

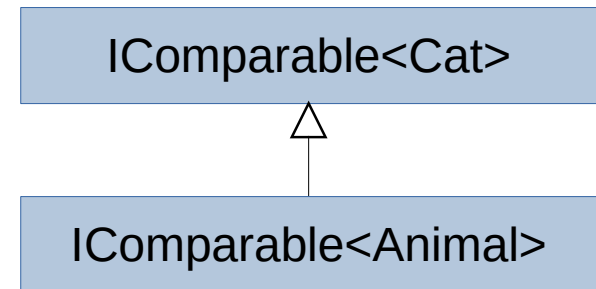
“in” keyword means:

T will only be used as input parameter



Contravariant type

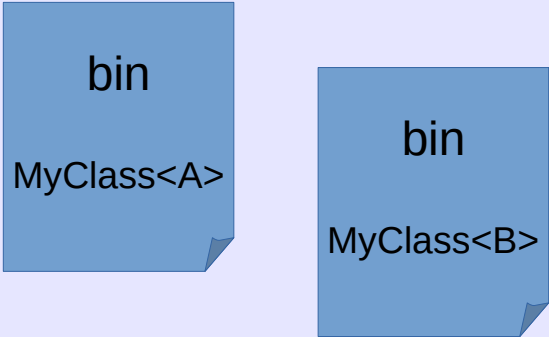
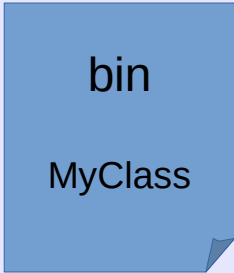

So:



Declaration-site (vs used-site in Java): Easier to use a component but harder to write its API



Generic programming implementations

C++ Template	Java Generics type erasure	C# Generics reification
<pre>template <typename T> class MyClass { T get() {...} } MyClass<A> ma; MyClass mb; A a = ma.get();</pre>	<pre>class MyClass<T> { T get() {...} } MyClass<A> ma; MyClass mb; // Compiler implicitly add // cast A a = (A) ma.get();</pre>	<pre>class MyClass<T> { T get() {...} } MyClass<A> ma; MyClass mb; A a = ma.get();</pre>
Generated at compile time	Only one class compiled	Generated at runtime
		
		Different binaries with primitive types

Reference

- <http://www.angelikalanger.com/GenericsFAQ/FAQSections/ParameterizedTypes.html>
- **A Conversation with Anders Hejlsberg (part VII)**
by Bill Venners with Bruce Eckel
<https://www.artima.com/intv/anders.html>
- <https://docs.oracle.com/javase/tutorial/java/generics/index.html>
- **Effective Java - Joshua Bloch**
- <https://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java>

