Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Generics (Part 2)

Object Oriented Programming

Tiziano Leidi

09.11.2021

Limits of Formal Type Parameters

In generic types and methods, formal parameters are very useful to parametrize the inputs and outputs: they are mainly used when declaring the parameters of constructors and methods, as well as the methods' return types.

However, they quickly show their weaknesses, when access to the referenced objects inside the generic types or methods is needed.

Limits of Formal Type Parameters

```
class GenType<T> {
   private T t;

   public T get() {
       return this.t;
   }

   public void set(T t) {
       this.t = t;
   }

   public void performOperation() {
       t.???
   }
}
```

Limits of Formal Type Parameters

When a generic class, interface or method is defined, the Java compiler has no notion about the concrete type that will be used at execution time, therefore all specific fields or methods of the concrete type are hidden, even if public, leaving available just the functionality of *Object*.

Bounded Type Parameters

For this reason, Java allows to specify upper bounds for the type of the formal parameters. Then, the type of the actual argument must be compatible with the specified upper-bound type.

To specify the upper bound, the reserved word "extends" is used (for classes, interfaces as well as for methods).

Bounded Type Parameters

```
class GenType<T extends Number> {
   private T t;

   public T get() {
       return this.t;
   }

   public void set(T t) {
       this.t = t;
   }

   public void performOperation() {
      double doubleValue = t.doubleValue();
      // ...
   }
}
```

Bounded Type Parameters

Upper-bounds provide information on the classes and interfaces with which the formal parameter is compatible.

As a result, public methods and fields of the upper-bound classes and interfaces become visible inside the generic type or method.

Multiple Bounds

It is possible to specify more than one upper-bound at the same time.

If one of the bounds is a class, it has to be specified for first.

```
class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

```
abstract class GeometricFigure {
    private int x = 0, y = 0;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

```
interface ShapeWithColor {
    int getColor();
}
```

```
public class Box<T extends GeometricFigure & ShapeWithColor> {
    private T t;

    // …

    @Override
    public String toString() {
        return "Geometric figure at x: " + t.getX() + " and y: " + t.getY()
                + " and color: " + t.getColor();
    }
}
```

Bounded Type Parameters in Methods

It is possible to use bounded type parameters also in generic methods.

For example, the following method counts the number of elements that are greater than a certain element. Where is the problem?

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)
            ++count;
    return count;
}
```

Bounded Type Parameters in Methods

Limited generic version:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generic Types and Inheritance

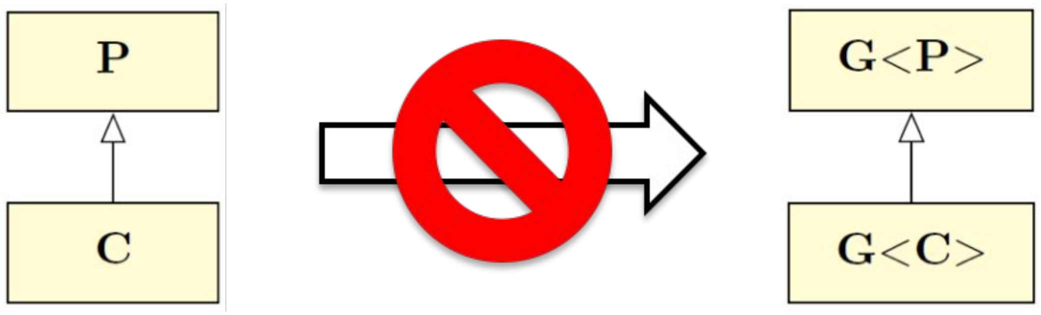
There are still some obstacles ...

Generic Types and Inheritance

Use of generics is subject to the following constraint:

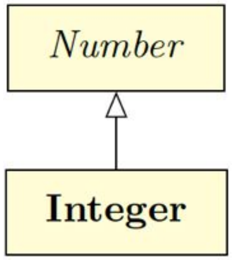
If C is a subtype of P (inherits from P), and G is a generic type statement, it does NOT automatically follows that:

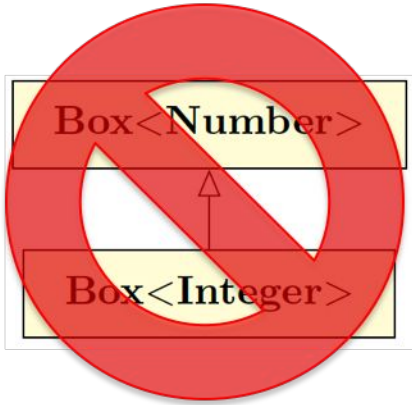
G<C> is a subtype of G<P>.



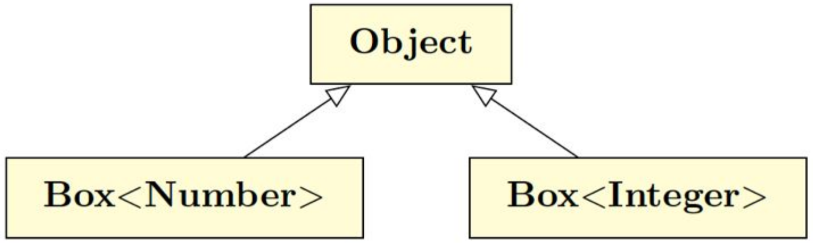
Generic Types and Inheritance

For example: even if Integer "is a" Number, Box<Integer> is not automagically subtype of Box<Number>.



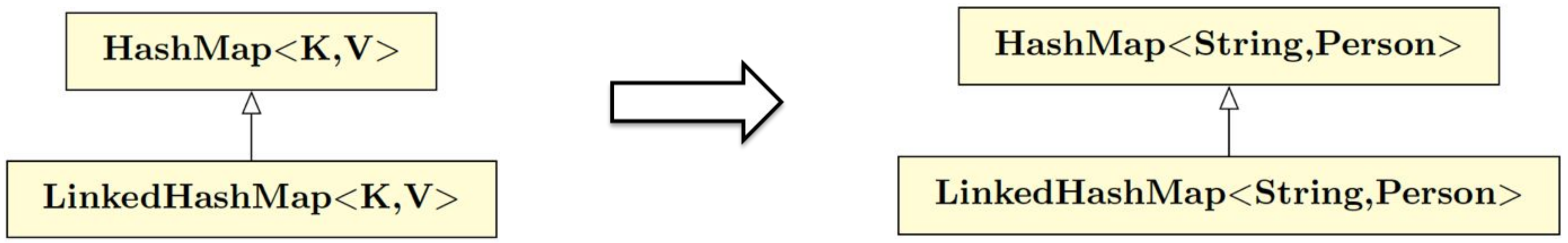


The only common compatible type is Object.



Generic Types and Inheritance

Inheritance relationships between generic types are preserved only if actual type arguments remain identical.



Generic Types and Inheritance

This constraint is of fundamental importance, otherwise it would be possible to perform an operation such as:

```
List<String> stringList = new ArrayList<>();
List<Object> objectList = stringList;
```

... then, by accessing the variable "objectList" any type of object could be added to the List of Strings, which is exactly what we want to avoid with Generics.

Wildcards

Therefore, if there is compatibility of formal type parameters, Java provides an additional tool to specify the presence of subtyping relationships between generic types: wildcards.

Wildcard are specified with the ? keyword.

Wildcards are used to declare unknown types.

```
public abstract class Shape {
    private int x, y;
    public abstract void draw();
}
```

```
public class Circle extends Shape {
    private int radius;
    @Override
    public void draw() {
        //...
    }
}
```

```
public class Rectangle extends Shape {
    private int width, height;
    @Override
    public void draw() {
        //...
    }
}
```

Without wildcard:

```
public void drawAll(List<Shape> shapes) {
    for (Shape s : shapes) {
        s.draw();
    }
}
```

!

```
List<Shape> shapes = new ArrayList<>();
List<Rectangle> rectangles = new ArrayList<>();
List<Circle> circles = new ArrayList<>();
// ...
drawAll(shapes);
```

~~drawAll(rectangles~~);

~~drawAll(circles~~);

Not allowed,
List<Rectangle>
and List<Circle>
are not
List<Shape>

With wildcard:

```
public void drawAll(List<? extends Shape> shapes) {
    for (Shape s : shapes) {
        s.draw();
    }
}
```

Wildcard

```
List<Shape> shapes = new ArrayList<>();
List<Rectangle> rectangles = new ArrayList<>();
List<Circle> circles = new ArrayList<>();
// ...
drawAll(shapes);
drawAll(rectangles);
drawAll(circles);
```

Wildcards

There are three types of wildcards:

- Upper-bounded wildcard
- Lower-bounded wildcard
- Unbounded wildcard

It is not possible to specify upper-bound and lower-bound simultaneously.

Upper-Bounded Wildcards

Syntax: *"? extends Type"*

Is the family of all subtypes of *Type*. It is used to express compatibility in a ways similar to standard type compatibility (polymorphism).
if *A* is any concrete type that extends *B*, and *G* a generic type, then *G<A>* is compatible with *G<? extends B>*.

Is the most used and useful wildcard. Mostly used for parameters that have an input behaviour.

Lower-Bounded Wildcards

Syntax: *"? super Type"*

Is the family of all supertypes of *Type*. It is used to express compatibility with types that are between Object and *Type*.
if *A* is any concrete type that extends *B*, and *G* a generic type, then *G*<*B*> is compatible with *G*<*? super A*>.

Mostly used for parameters that have an output behaviour.

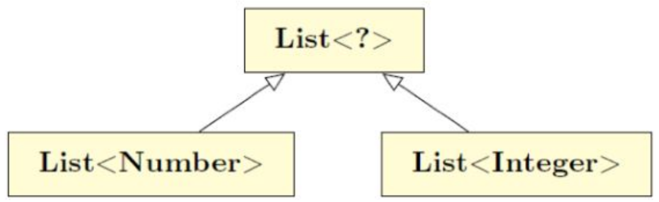
Unbounded Wildcards

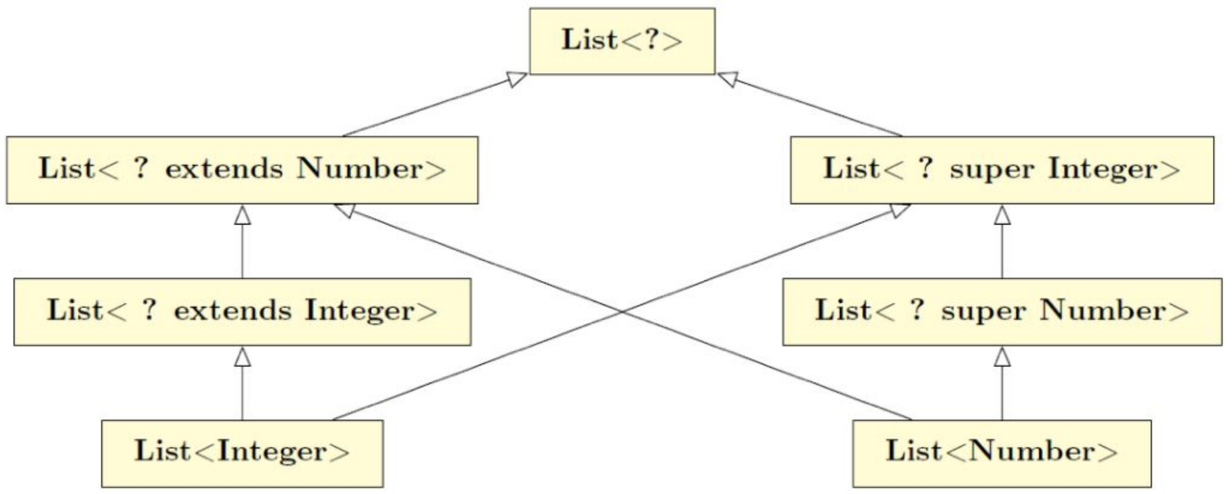
Syntax: *"?"*

Represent all types. It is compatible with everything, but it's limited in terms of supported operations. Should be avoided.

if *A* is any concrete type, and *G* a generic type, then *G*<*A*> is compatible with G<?>. For example, the supertype of all types of collections is: Collection<?>

Wildcards


List<?>
List<Number>
List<Integer>


List<?>
List< ? extends Number>
List< ? super Integer>
List< ? extends Integer>
List< ? super Number>
List<Integer>
List<Number>

Supported Operations

If you use wildcards, the compiler blocks some of the possible operations.

For example it is not possible to do:

```
public void addRectangle(List<? extends Shape> shapes) {
    // Compile-time error!
    shapes.add(0, new Rectangle());
}
```

… because the type of the list is unknown. Any List of Shape either compatible or not with Rectangle, could be provided as argument.

Supported Operations

Unbounded wildcards are rarely used, because all operations are blocked. May be useful in exceptional cases:

- when the generic type or method does not directly depend on the formal parameter
- when the generic type or method only accesses functionality from the Object class

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

```
class A {
}

class B extends A {
}

class C extends B {
}

class D extends B {
}
```

```
public class WildCard {
    public static void main(String[] args) {
        List<? extends A> la;
        la = new ArrayList<B>();
        la = new ArrayList<C>();
        la = new ArrayList<D>();
        List<? super B> lb;
        lb = new ArrayList<A>(); // fine
        lb = new ArrayList<C>(); // will not compile
    }

    public void someMethod(List<? extends B> lb) {
        B b = lb.get(0); // is fine
        lb.add(new B()); // will not compile as we do not know the type of the
                          // list, only that it is bounded above by B
    }

    public void otherMethod(List<? super B> lb) {
        B b = lb.get(0); // will not compile as we do not know whether the
                         // list is of type B, it may be a List<A> and
                         // only contain instances of A
        lb.add(new B()); // is fine, as we know that it will be a super type of B
    }
}
```

Standard Use of Wildcards

Therefore, in normal situations:

- Upper-bounded wildcards are used if the generic type has input behaviour (to allow read of values)
- Lower-bounded wildcards are used if the generic type has output behaviour (to allow write of values)
- Avoid using wildcards if both input and output behaviour is expected.

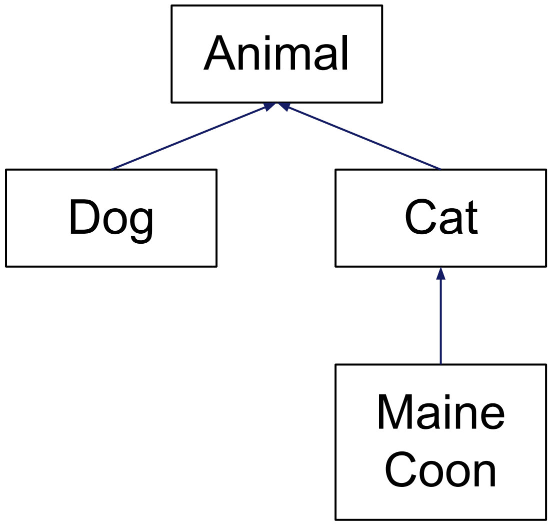
Example

```
public class Animal {
    private String name;

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

    public String getName() {
        return name;
    }

    public void jump() {
        System.out.println(getName() + " is jumping.");
    }
}
```



Example

```
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
    public void bark() {
        System.out.println(getName() + " is barking.");
    }
}
```

```
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }
    public void meow() {
        System.out.println(getName() + " is meowing.");
    }
}
```

```
public class MaineCoon extends Cat {
    public MaineCoon(String name) {
        super(name);
    }
    public void meow() {
        System.out.println("Maine Coon" + getName()
                                    + " is meowing.");
    }
}
```

Example

```java
public class AnimalTrainer {
    public void act(List<Animal> animalList) {
        for (Animal animal : animalList)
            animal.jump();
    }
}
```

```java
public class Circus1 {
    public static void main(String[] args) {
        AnimalTrainer animalTrainer =
                new AnimalTrainer();
        List<Animal> animalList =
                new ArrayList<>();
        animalList.add(new Dog("Bob"));
        animalList.add(new Dog("Amy"));
        animalTrainer.act(animalList);
    }
}
```

```java
public class Circus2 {
    public static void main(String[] args) {
        AnimalTrainer animalTrainer =
                new AnimalTrainer();
        List<Cat> catList =
                new ArrayList<>();
        catList.add(new Cat("Tim"));
        catList.add(new Cat("Lucy"));
        animalTrainer.act(catList); // ERROR!!!
    }
}
```

Example

```
public class AnimalTrainer {
    public void act(List<? extends Animal> animalList) {
        for (Animal animal : animalList)
            animal.jump();
    }
}
```

```
public class Circus {
    public static void main(String[] args) {
        AnimalTrainer animalTrainer = new AnimalTrainer();
        List<Cat> catList = new ArrayList<>();
        catList.add(new Cat("Tim"));
        catList.add(new Cat("Lucy"));
        animalTrainer.act(catList); // NO ERROR!!!
    }
}
```

```
public class CatTrainer {
    public void act(List<? extends Cat> catList) {
        for (Cat cat : catList)
            cat.meow();
    }
}
```

Example

```
public class DogKeeper {
    public void addDogs(List<? super Dog> animalList) {
        animalList.add(new Dog("Bob"));
        animalList.add(new Dog("Amy"));
    }
}
```

```
public class CatKeeper {
    public void addCats(List<? super Cat> animalList) {
        animalList.add(new Cat("Tim"));
        animalList.add(new Cat("Lucy"));
        animalList.add(new MaineCoon("Sally"));
    }
}
```

Example

```
public class Circus3 {
    public static void main(String[] args) {
        List<Animal> animalList = new ArrayList<>();
        DogKeeper dogKeeper = new DogKeeper();
        dogKeeper.addDogs(animalList);
        CatKeeper catKeeper = new CatKeeper();
        catKeeper.addCats(animalList);
        AnimalTrainer animalTrainer = new AnimalTrainer();
        animalTrainer.act(animalList);

        List<Cat> catList = new ArrayList<>();
        catKeeper.addCats(catList);
        CatTrainer catTrainer = new CatTrainer();
        catTrainer.act(catList);
    }
}
```

Summary

- Limits of formal parameter types
- Bounded type parameters
- Inheritance implications of generic types
- Wildcards