

Generics

by: Tamas Marincsak



Table of Contents

Show me the money!

Introduction

Generic class

Raw types

Type parameter

Type parameter bounding

Wildcard argument

Wildcard argument bounding

Generic method

Generic constructor

Inheritance

Generic interface

Type inference

Ambiguity error


Some restrictions

Summary

marinchuck/
generics




 0
Contributor

 0
Issues

 1
Star

 0
Forks





What do we win with generics?

- no need for ~~explicit casting~~
- type checking in **compile time** (~~ClassCastException~~)
- (enabling programmers to implement **generic algorithms**)



Introduction

- introduced by *JDK 5*
- new syntactic elements (?, <>)
- rewrite core Java classes, methods, etc.
- can be ***class, interface, method, constructor***

1. Class

A class is generic if it declares one or more type variables. These type variables are known as the type parameters.

syntax: `class class-name<type-param-list> {...}`

type parameter scope: inside class definition block

- **type erasure:**

To implement generics and to also preserve backward compatibility *javac* applies type erasure.

- replaces all type parameters with their bounds or with `Object` if the parameter is unbounded
- inserts type casts if necessary
- generates bridge methods to preserve polymorphism

```
1 class Gen<T> {
2
3     private T ob;
4
5     public Gen(T o) {
6         ob = o;
7     }
8
9     public T getOb() {
10         return ob;
11     }
12
13     public void setOb(T o) {
14         this.ob = o;
15     }
16 }
```

type erasure
↔

```
1 class Gen {
2
3     private Object ob;
4
5     public Gen(Object o) {
6         ob = o;
7     }
8
9     public Object getOb() {
10         return ob;
11     }
12
13     public void setOb(Object o){
14         this.ob = o;
15     }
16 }
```

Raw types

- support for generics did not exist prior to *JDK 5*
- it was necessary to provide a transition path which enables pre-generics code to remain functional while at the same time being compatible with generics.
- to handle the transition to generics, Java allows a generic class to be used without any type arguments, resulting in **raw type**
- **disadvantage:**
type safety of generics is lost

```
1 {  
2     List myList = new ArrayList();  
3  
4     myList.add("one");  
5  
6     for(int i=0; i<myList.size(); i++){  
7         //explicit type cast needed  
8         String str = (String) myList.get(i);  
9         System.out.println("The "+i+"th element is: "+str);  
10    }  
11 }
```

Type parameter

- scope depends on where it is declared
- needs to be declared inside `<>`
- no special significance to the name T
- naming conventions:
 - E: Element (mainly used by *Collection Framework*)
 - K: Key
 - N: Number
 - T: Type
 - V: Value
 - S, U, etc.

```
1 class Gen<T> {  
2  
3     private T ob;  
4  
5     public Gen(T o) {  
6         ob = o;  
7     }  
8  
9     public T getOb() {  
10        return ob;  
11    }  
12  
13    public void setOb(T o) {  
14        this.ob = o;  
15    }  
16 }
```


Type parameter

- *JDK 10*: cannot be **var**
- arguments can only be reference types
eg.: `Integer`, `List<Integer>`, `int[]`
- more than one type parameter in a comma-separated list

```
1 class Gen<T, V> {
2
3     T obT;
4     V obV;
5 }
6
7 {
8 //      T      V      T      V
9     Gen<Integer, String> genReference = new Gen<Integer, String>(69, "is awesome!");
10    Gen<String, String> x = new Gen<String, String>("69", "is awesome");
11 }
```

Type parameter - Bounding

We can use inheritance in order **to limit** the types that can be substituted.

syntax: `<T extends superclass>`

This specifies that T can only be replaced by superclass, or subclasses of superclass, thus superclass defines an inclusive, upper limit.

- limit can be an *interface* or a *class*
- only *upper* bound
- only specified types can be extended, else you have to specify it by listing it as a type parameter

```
1 //valid
2 class Box<U extends Number> {}
3
4 //invalid
5 class Box<U extends V> {}
6
7 //valid
8 class Box<U extends V, V> {}
```

Type parameter - Bounding

- a type parameter can have multiple bounds
 - & operator
 - only 1 class
 - class has to come first
- if the superclass is generic, and it doesn't have a specified argument, than it's type has to be listed in the parameter list

```
1 class A {}
2
3 interface B {}
4
5 interface C {}
6
7 //valid
8 class D <T extends A & B & C> {}
9
10 //invalid
11 class D <T extends B & A & C> {}
12
13
14 //valid
15 class A<T extends B<String>> {}
16
17 //valid
18 class A<T extends B<T>> {}
19
20 //invalid
21 class A<T extends B<U>> {}
22
23 //valid
24 class A<T extends B<U>, U> {}
```

Wildcard argument

In generic code, the question mark (?), called the **wildcard**, represents an **unknown type**.

- **can** use a generic type with a wildcard:
 - as a parameter in a method
 - as a field
 - as a local variable
- **can't** use a generic type with a wildcard:
 - as a type argument for a generic method invocation
 - at generic class instance creation
 - as a supertype

```
1 class A {
2     //valid
3     private List<?> list;
4
5     //valid
6     public List<?> getList(){
7         return this.list;
8     }
9
10    public void setList(List<?> list){
11        //valid
12        List<?> myListCopy = list;
13        this.list = list;
14    }
15 }
16
17 {
18     //invalid
19     Arrays.<?>asList(6, 9, "=69");
20
21     //invalid
22     List<String> list = new ArrayList<?>();
23
24     //invalid
25     class MyList implements List<?> {}
26 }
```

Wildcard argument - Bounding

We can use inheritance in order **to limit** the types that can be substituted.

- limit can be an *interface* or a *class*
- works only with specified types
- no multiple bounds
- upper bound ***and*** lower bound

```
1 {  
2     //invalid  
3     List<? extends A & B> foobar;  
4     A & B item = foobar.get(0)  
5 }
```

Wildcard argument - Bounding

- *Upper:* **syntax: `<? extends superclass>`**

This specifies, that ? can only be superclass type, or a type that is a subclass of the superclass, thus superclass defines an inclusive, upper limit.

- *Lower:* **syntax: `<? super subclass>`**

This specifies, that ? can only be subclass type, or a type that is a superclass of the subclass, thus subclass defines an inclusive, lower limit.

```
1 {
2  /*
3   * Suppose we want to accept lists, which can contain any Number type
4   * eg.: List<Integer>, List<Double>...etc.
5   */
6   public void printList(List<Number> list){} //not OK
7   public void printList(List<? extends Number> list){} //OK
8
9   /*
10  * Suppose we want to accept lists, which can contain Integer type
11  * eg.: List<Integer>, List<Number>, List<Object>...etc.
12  */
13  public void printList(List<Integer> list){} //not OK
14  public void printList(List<? super Integer> list){} //OK
15 }
```


2. Method

A method is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the method.

/Generic methods are methods that introduce their own type parameters./

syntax: `<formal-type-param-list > ret-type meth-name (param-list) {...}`

formal type parameter scope: inside method definition block

- same type parameter rules apply
- can be declared in generic and in non-generic class or interface
- you can explicitly specify the type argument if needed on calling:

```
StringLengthChecker.  
<Integer, String>compareStringLength(69, "ok")
```

```
1 class StringLengthChecker {  
2     public static <K, V> int compareStringLength(K ob1,V ob2){  
3         Integer ob1Length=ob1.toString().length();  
4         Integer ob2Length=ob2.toString().length();  
5         return ob1Length.compareTo(ob2Length);  
6     }  
7 }
```

3. Constructor

A constructor is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the constructor.

/Generic constructors are constructors that introduce their own type parameters./

syntax: `<formal-type-param-list > constructor-name (param-list) {...}`

formal type parameter scope: inside constructor definition block

- same type parameter rules apply
- can be declared in generic and in non-generic class
- you cannot explicitly specify the type argument on calling

```
1 class A {  
2  
3     private double val;  
4  
5     <T extends Number> A(T arg) {  
6         val = arg.doubleValue();  
7     }  
8  
9     void showVal() {  
10         System.out.println("val: " + val);  
11     }  
12 }
```


Inheritance

In a generic hierarchy, any type arguments needed by a generic superclass or interface must be passed up the hierarchy by all subclasses.

- only the not specified type arguments have to be passed!
- if the needed type argument is a bounded type, then the inheritor has to define its bound.
Once this bound has been established, its not allowed to specify it again!

```
1 class Gen<T> {}
2
3 interface GenI<T> {}
4
5 class Gen2<U, V> extends Gen<U> implements GenI<V> {}
6
7 class Gen3 extends Gen<String> {}
8
9
10 interface GenI<T extends Number> {}
11
12 //valid
13 class Gen<U extends Number> implements GenI<U> {}
14
15 //valid
16 class Gen2<U extends Integer> implements GenI<U> {}
17
18 //invalid
19 class Gen3<U extends Integer> implements GenI<U extends Integer> {}
```

4. Interface

An interface is generic if it declares one or more type variables. These type variables are known as the type parameters of the interface.

syntax: `interface interface-name<type-param-list> {...}`

type parameter scope: inside interface definition block

- same type parameter rules apply

```
1 public interface List<E> extends Collection<E> {}
```

Type inference

Type inference is the ability of the Java compiler to infer datatypes based on corresponding declarations.

- *JDK 7* introduced the diamond operator `<>`

- prior to JDK 7: `class-name<type-arg-list> var-name = new class-name <same-type-arg-list>(cons-arg-list);`

- after JDK 7: `class-name<type-arg-list > var-name = new class-name <>(cons-arg-list);`

- *JDK 10* introduced the `var` keyword

- after JDK 10: `var var-name = new class-name <type-arg-list>(cons-arg-list);`

```
1 {
2   Map<Integer, String> myMap=new HashMap<Integer, String>();
3   Map<Integer, String> myMap=new HashMap<>();
4   var myMap=new HashMap<Integer, String>();
5 }
```

Ambiguity error

Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict, resulting a compile time error.

- mainly happens on **overloading**
(same name, same class or interface, different signature)

```
1  class MyGenClass<T, V> {
2
3      T ob1;
4      V ob2;
5
6      // These two overloaded methods are ambiguous
7      // and will not compile.
8      void set(T o) {
9          ob1 = o;
10     }
11
12     void set(V o) {
13         ob2 = o;
14     }
15 }
```

type erasure
↔
= compile-time error!

```
1  class MyGenClass {
2
3      Object ob1;
4      Object ob2;
5
6      void set(Object o) {
7          ob1 = o;
8      }
9
10     void set(Object o) {
11         ob2 = o;
12     }
13 }
```

Some restrictions

- cannot instantiate a **type parameter**
- cannot instantiate an **array** whose element type is a *type parameter*
- cannot create an **array** of *type-specific generic references*

```
1 class Gen<T> {  
2     T ob;  
3     Gen() {  
4         //invalid  
5         ob = new T();  
6     }  
7 }
```

```
1 class Gen<T> {  
2     T[] vals;  
3     Gen() {  
4         //invalid  
5         vals = new T[10];  
6     }  
7 }
```

```
1 //invalid  
2 List<String>[] myStringListArray = new List<>[10];  
3 //valid  
4 List<?>[] myStringListArray = new List<?>[10];
```

Some restrictions

- **no static member** can use a *type parameter* declared by the enclosing class
- a generic class cannot extend **Throwable**, meaning you cannot create generic exception classes

```
1 class Gen<T> {  
2  
3     //invalid  
4     static T ob;  
5  
6     //invalid  
7     static T getOb() {  
8         return ob;  
9     }  
10 }
```

```
1 //invalid  
2 class Gen<T> extends Throwable {}
```

Summary

introduced by JDK 5, no explicit type-casting, type check at compile-time, can be class | interface | method | constructor, type erasure, raw types, type parameter and it's rules, type parameter bounding, wildcard argument (?), wildcard argument and it's bounding, inheritance between generics, type inference (<> JDK7, var JDK10), ambiguity compile time error, some restrictions

- *further reading: bridge methods*

The End