Generics

J2SE5

Introduction

- “*GENERIC*” means “Parameterized types”
- Starting with version 5, java allows class definitions that contains a parameter (or parameters) for a type or types
- Classes and methods can have type parameters
- The type parameter then may have any reference type, and hence, any class type, plugged in for the type parameter

A Simple Generic Class

```
class Gen<T>
{
    T ob;
    Gen(T o){
        ob=o;
    }

    T getOb() {
        return ob;
    }
    //show type of T
    public void showType() {
        System.out.println("Type of T is "+ob.getClass().getName());
    }
}
```

A Simple Generic Class

```
class GenericsDemo1
{
	public static void main(String[] args)
	{
		Gen<Integer> gi = new Gen<Integer>(66); //autoboxing!!!
		gi.showType();
		int i =gi.getOb();//autounboxing!!

		System.out.println(i);

		Gen<String> gs = new Gen<String>("Generics");
		gs.showType();
		String s =gs.getOb();//autounboxing!!

		System.out.println(s);
	}
}
```

Using Generics

- You may use any non-keyword identifier for the type parameter
- By convention a type parameter starts with an Uppercase letter
- It is Tradition to use a single letter

A generic class With two type parameters

```
class TwoGen <T,V>
{
        T ob1;
        V ob2;
        TwoGen(T o1, V o2)
        {
                ob1=o1;
                ob2=o2;
        }
        void showTypes() {
                System.out.println("Type of T is "+ob1.getClass().getName());
                System.out.println("Type of T is "+ob2.getClass().getName());
        }

        T getOb1() {
                return ob1;
        }

        V getOb2() {
                return ob2;
        }
};
```

A generic class With two type parameters

```
class TwoGenDemo
{
    public static void main(String[] args)
    {
    TwoGen<Integer, String> tgObj = new TwoGen(Integer,String)(88, Generics);
          tgObjg.showTypes();

          int v = tgObj.getOb1();
          String str = tgObj.getOb2();
    }
};
```

Bounded parameters

- Let’s consider an example where we have two elements in a pair
- We want to add a method that returns the maximum of the two values.
- Here the “maximum is relative”!!!

Bounded parameters

```
class Pair<T>
{
    private T first;
    private T second;
    if (first.compareTo(second))
                     return first;
    else
            return second;
};
```

Which interface do we need to implement ?

Comparable

Bounded parameters

```
class Pair<T extends Comparable>
{
    private T first;
    private T second;
    if (first.compareTo(second))
                    return first;
    else
            return second;
};
```

More On Bounds

- A bound on a Type may be a class (rather than an interface)
- In this case only the descendant classes of the bounding class may be plugged in for the type parameter

```
public class SomeClass<T extends A>
```

One More Example on Bounds

```
class Stats<T>
{
      T[ ] nums;
      Stats(T[ ] o)
      {
                nums=o;
      }
      double average() {
                double sum=0;
                          for (int i=0;i<nums.length ;i++ )
                          {
                                    sum+=nums[i].doubleValue()
                          // doubleValue() method cannot be used for all Types!!
                          }
                          return sum;
      }
};
```

When you try to compile it , an error is reported saying doubleValue() is not found. This problem can be solved by mentioning the type parameter for Stats as <T extends Number> indicating that all types must be a subtype of Number class as Number defines doubleValue();

One More Example on Bounds

```
class Stats<T extends Number>
{
    T[ ] nums;
    Stats(T[ ] o)
    {
          nums=o;
    }
    double average() {
          double sum=0;
                    for (int i=0;i<nums.length ;i++ )
                    {
                              sum+=nums[i].doubleValue()
                    // doubleValue() method cannot be used for all Types!!
                    }

                    return sum;
}
};
```

Using Wildcards(?) – the Problem

- Using the previous example we can create two instances of stats and calculate averages as:

```
Integer inums ={1,2,3,,4,5};

Double dnums={1.1,2.2,3.3,4.4,5.5};
```

- Now let us check whether their averages are Equal? As

```
If(iob.sameAgerage(dob)) //to define

  System.out.println(“Same average);

    else

  System.out.println(“Not Same average);
```

Using Wildcards(?)

- Lets us define “sameAverage(..)

```
boolean sameAverage(Stats<T> ob)
    {
        if(average()==ob.average())
                return true;
        else
                return false;
    }
```

- Does it work?
- What do you specify for Stat’s type parameter when you declare a parameter of that type (Stat<Stat> ???!!!)
- If the invoking Object is of type Stat<Integer>, then the parameter “ob “ must be of type Stat<Integer>
- It can’t be used to compare the average of an object of type Stats<Double> with an Object of Stats<Short>

Using Wildcards(?) – The Solution

- To create a Generic “sameAverage” method a ? (wild card) argument is passed in the declaration

```
boolean sameAverage(Stats<?> ob)
    {
        if(average()==ob.average())
                return true;
        else
                return false;
    }
```

  - Here Stats<?> matches any Stats object, allowing any two objects to have their average compared.
  - The wild card parameter can be bounded also.
    - <? extends superclass>

Generic Methods

- You can also define a generic method that has its own type parameter that is not type parameter of any class.
- This generic method can be a member of an ordinary (i.e., non-generic) class…
- … or a member of some generic class with some other type parameter.

Generic Methods

Definition:

```
public static <T, V extends T> boolean isIn(t x, V[] y)
```

Or

```
public static <T> T getFirst(T[ ] x){
Return x[0];
}
```

- **Note** the generic Type declaration is placed AFTER all the modifiers and before the RETURN Type

Generic Constructors

- Constructors could be generic even if the class is not Generic

```
class GenCons {
private double v;
    <t extends Number> GenCons(T arg) {
                    val = args.doubleValue();
          }
    void showVal() {
                    System.out.println(“ Val : “+val);
          }
}
```

Generic Interface

- Declaration

```
public interface MinMax <T> {
   //Indicates this interface can be applied for only of
   //objects of type T
}
public interface MinMax <T extends Comparable> {
//this interface can be used only for those objects
//which can be ordered
}
```

Implementing a Generic Interface

```
class MyClass< T extends Comparable>
   implements MinMax<T>{

}
```

- The Type parameter is declared by “MyClass” and passed to MinMax.
- As MinMax requires a type that extends Comparable, the implementing class (MyClass) must specify the **same** bound.
- Once the bound has been established , there is no need to specify it again in the implements clause