Generic class

Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

Example to use Generics in java

```
ArrayList<String>
```

Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let’s see the simple example to create and use the generic class.

Creating generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let’s see the code to use the generic class.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

```
Output:2
```

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type

2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let’s see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```java
public class TestGenerics4{

   public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
         }
         System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Test it Now

```
Output:Printing Integer Array
      10
      20
      30
      40
```

```
50
Printing Character Array
J
A
V
A
T
P
O
I
N
T
```

But what to create custom generic classes?

Consider a Library. We can have a book library of audio library or cd library, whatever.Library class is.

```
package com.thejavageek.generics;

import java.util.List;

public class Library {

        private List items ;

        public Library(List items){
                this.items = items;
        }

        public Object issueItem(){
```

```
        // write code to issue item.
        return items.get(0);
    }

}
```

Now if we want to make a library of books, then we need to extends this class.

```
package com.thejavageek.generics;

import java.util.List;

public class BookLibrary extends Library {

    public BookLibrary(List<Book> items) {
        super(items);
    }

    public Object issueItem() {
        return super.issueItem();
    }

    public void returnItem(Object item) {
        if (item instanceof Book) {
            super.returnItem(item);
        } else {
            // throw some exception as can't add anything other than Book
        }

    }

}
```

We created BookLibrary but there are a lot of issues here.

- We have to create a new class just to make it a library of books.
- We have to check the type of Item in returnItem method as the overriding method parameter type must be that of superclass method.
- What we we need to make libraries of books, cds, news papers, magazines and lots of them. Are we going to create a separate class for that? Of course not.

What we need is a generic way to do this. We need to create a custom generic class.

So we will create a generic library like this.

```
package com.thejavageek.generics;

import java.util.List;

public class Library<T> { // "T" is the Type parameter. We can create Library of any Type

    private List<T> items ; // represents the list of items of type we will pass

    public Library(List<T> items){
        this.items = items;
    }

    public Object issueItem(){
        // write code to issue item.
        return items.get(0);
    }

    public void returnItem(T item){ // again pass the same "T" type to issueItem method
        items.add(item);
    }

}
```

Let us create Library now

```
package com.thejavageek.generics;

import java.util.ArrayList;
import java.util.List;

public class TestLibrary {

    public static void main(String[] args) {

        List<Book> books = new ArrayList<Book>();
        books.add(new Book());
        books.add(new Book());
        books.add(new Book());

        Library<Book> bookLibrary = new Library<Book>(books);

        Book book = bookLibrary.issueItem();
        System.out.println(book);
        List<CD> cds = new ArrayList<CD>();
        cds.add(new CD());
        cds.add(new CD());
        cds.add(new CD());

        Library<CD> cdLibrary = new Library<CD>(cds);

        CD cd = cdLibrary.issueItem();
        System.out.println(cd);
    }

}
```

This outputs,

```
com.thejavageek.generics.Book@45bab50a
com.thejavageek.generics.CD@7150bd4d
```

This means we can successfully create a generic type which can type any Type as argument and we can perform operations on it. We do not have to write Type specific methods. Generics handles everything.

Wildcard in Java Generics

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

```
import java.util.*;
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}


class GenericTest{
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){
s.draw();//calling method of Shape class by child class instance
}
}
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());

List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());

drawShapes(list1);
drawShapes(list2);
}}
```

```
drawing rectangle
drawing circle
drawing circle
```

http://www.thejavageek.com/2013/08/28/generics-the-wildcard-operator/

In the previous part of this tutorial, Polymorphism with generics we have learned the shortcomings of generics while using polymorphism with it.In this part, we are going to overcome them using wildcard operator.

Please read this article before continuing.

Vehicle Class

```
package com.thejavageek.generics;

public class Vehicle {

        public void service() {
                System.out.println("Generic vehicle servicing");
        }

}
```

Bike Class

```
package com.thejavageek.generics;
```

```java

public class Bike extends Vehicle {
    @Override
    public void service(){
        System.out.println("Bike specific servicing");
    }
}
```

Car class

```java
package com.thejavageek.generics;

public class Car extends Vehicle {

    @Override
    public void service() {
        System.out.println("Car specific servicing");
    }

}
```

Mechanic Class

```java
package com.thejavageek.generics;
```

```

import java.util.ArrayList;
import java.util.List;

public class Mechanic {

    public void addVehicle(List<Vehicle> vehicles){
        // some code here
    }

    public static void main(String[] args) {

        List<Vehicle> vehicles = new ArrayList<Vehicle>();
        vehicles.add(new Vehicle());
        vehicles.add(new Vehicle());

        List<Bike> bikes = new ArrayList<Bike>();
        bikes.add(new Bike());
        bikes.add(new Bike());

        List<Car> cars = new ArrayList<Car>();
        cars.add(new Car());
        cars.add(new Car());

        Mechanic mechanic = new Mechanic();

        mechanic.addVehicle(vehicles); // compiles fine
        mechanic.addVehicle(bikes);    // compilation fails
        mechanic.addVehicle(cars);     // compilation fails

    }

}
```

Now we want to be able to pass a ArrayList<Bike> to addVehicle(List<Vehicle> vehicles)successfully but due to type erasure, as discussed in previous article, it is not possible. Here comes the use of wildcard operator i.e. ? .

Wildcard operator and extends keyword

We will change the method to

```
public void addVehicle(List<? extends Vehicle> vehicles){
      // some code here
}
```

After doing this,

```
mechanic.addVehicle(vehicles); // compiles fine
mechanic.addVehicle(bikes);    // compiles fine
mechanic.addVehicle(cars);     // compiles fine
```

So what does it mean ?

The statement List<? extends Vehicle> **means that you can pass any type of list that subtype of List and which is typed as Vehicle or some subtype of Vehicle. But you cannot add anything into the collection at all.**

Now what is that? Why can’t we add anything into the collection? Lets consider this.

```
public void addVehicle(List<? extends Vehicle> vehicles){
            vehicles.add(new Car());
}
```

The compiler stops you at the very moment. It doesn’t allow you to add anything while using ?extends. Why is that?

- ? extends allows us to pass any collection that is subytype of the method parameter which is typed as generic type of subtype of the generic type. i.e. we can pass an ArrayList<Vehicle> ,ArrayList<Bike> or ArrayList<Car> to it.
- But consider a scenario in which we are passing ArrayList<Bike> to addVehicle(List<?extends Vehicle>). In that case, if compiler didn’t stop us from adding into the collection, we would have added a Car into ArrayList<Bike>.

This is the scenario so ? extends doesn’t allow you to add into collection. But there is also a workaround for this. We can also pass a subtype collection and still be able to add into collection. We have to use the super keyword along with wildcard operator

The wildcard operator and super keyword.

Now change as follows

```
public void addVehicle(List<? super Bike> vehicles){
            vehicles.add(new Car());    // can't add a Car to list of Bike
            vehicles.add(new Bike());   // compiles fine because we are adding a Bike
            vehicles.add(new Vehicle());// can't add a Vehicle to a list of Bike
}
```

and we call this method using

```
mechanic.addVehicle(vehicles); //compiles fine
mechanic.addVehicle(bikes);    //compiles fine
mechanic.addVehicle(cars);     // does not compile
```

Now what is this?

What does List<? super Bike> mean?

You can pass any collection that is of type Bike or any super type of Bike**. i.e.** Vehicle**. and you can add elements into it. So Even if you pass a list of bikes or list of vehicles, you will still be able to add a** Bike **into it. But you cannot add a** Car **into the collection because our method parameter declaration allows only a list of bikes or list of supertype of** Bike, **i.e.**Vehicle**.**