//Polymorphism with Array and Generics :

```java
//Polymorphism with array

import java.util.*;

abstract class Animal
{
    public abstract void checkup();
}

class Dog extends Animal
{
    @Override
    public void checkup()
    {
        System.out.println("Dog checkup");
    }
}

class Cat extends Animal
{
    @Override
    public void checkup()
    {
        System.out.println("Cat checkup");
    }
}

class Bird extends Animal
{
    @Override
    public void checkup()
    {
        System.out.println("Bird checkup");
    }
}


public class  Test8
{
    public static void checkAnimals(Animal ...animals)
    {
        for(Animal animal : animals)
        {
            animal.checkup();
        }
    }

    public static void main(String[] args)
    {
        Dog []dogs={new Dog(), new Dog()};

        Cat []cats={new Cat(), new Cat(), new Cat()};

        Bird []birds = {new Bird(), new Bird()};


        checkAnimals(dogs);
        checkAnimals(cats);
        checkAnimals(birds);
    }
}
```

Note :-From the above program it is clear that polymorphism(Upcasting) concept works with array.

```java
import java.util.*;
abstract class Animal
{
    public abstract void checkup();
}

class Dog extends Animal
{
   @Override
    public void checkup()
    {
        System.out.println("Dog checkup");
    }
}

class Cat extends Animal
{
    @Override
    public void checkup()
    {
        System.out.println("Cat checkup");
    }
}
class Bird extends Animal
{
    @Override
    public void checkup()
    {
        System.out.println("Bird checkup");
    }
}
public class Test9
{

    public void checkAnimals(List<Animal> animals)
    {
        for(Animal animal : animals)
        {
          animal.checkup();
        }
    }

    public static void main(String[] args)
    {
        List<Dog> dogs = new ArrayList<>();
        dogs.add(new Dog());
        dogs.add(new Dog());

        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat());
        cats.add(new Cat());

        List<Bird> birds = new ArrayList<>();
        birds.add(new Bird());
        birds.add(new Bird());

        Test9 t = new Test9();
        t.checkAnimals(dogs);
        t.checkAnimals(cats);
        t.checkAnimals(birds);

    }
}
```

Note :- The above program will generate compilation error.

So from the above program it is clear that polymorphism does not work in the same way for generics as it does with arrays.

Example :

```java
Parent [] arr = new Child[5]; //valid
Object [] arr = new String[5]; //valid
```

But in generics the same type is not valid

```java
List<Object> list = new ArrayList<Integer>(); //Invalid
List<Parent> mylist = new ArrayList<Child>(); //Invalid
```

```java
import java.util.*;
public class Test10
{
public static void main(String [] args)
    {
       /*ArrayList<Object> al = new ArrayList<String>(); [Compile time ]
        ArrayList al = new ArrayList();  [Runtime, Type Erasure]
       al.add(true); */


         Object []obj = new String[3]; //valid with Array
         obj[0] = "Ravi";
         obj[1] = "hyd";
         obj[2] =  90; //java.lang.ArrayStoreException
         System.out.println(Arrays.toString(obj));
    }
}
```

Note :- Program will generate java.lang.ArrayStoreException because we are trying to insert 90 (integer value) into String array.

In Array we have an Exception called ArrayStoreException (Which protect us to assign some illegal value in the array at runtime) but the same Exception or such type of exception, is not available with Generics (due to Type Erasure) that is the reason in generics, compiler does not allow upcasting concept.
(It is a strict compile time protection)

WildCard (?)
----
* It is called Wild Card.

* We have following different cases with Wild Card.

Case 1 :

<Animal>   : Only Animal type is allowed.

Case 2 :

<Dog>   : Only Dog type is allowed.

Case 3 :

<?>   : It is unknown type so many possibilities that means It can accept anything.

Now, to provide some boundation we have 2 different cases :
a) Upper bound
b) Lower bound

Case 4 :
----

a) Upper Bound :

<? extends Animal> :

This is upper bound here we can replace wild-card (?) with any class which extends from Animal but here there is a chance of wrong collection, because in future Animal can have more sub classes so, we can't add any element in the Collection.

b) Lower Bound :

<? super Dog> :

This is called lower bound, Here we can replace wild card (?) with any class which is super of Dog i.e Animal, Object. Here compiler knows that the only classes which are super of Dog are allowed so adding element in the collection is allowed.

//Program :

```java
import java.util.*;

class Animal
{

}

class Dog extends Animal
{
}

class Horse extends Animal
{
}

public class WildCardDemo1
{
    public static void main(String[] args)
    {
       ArrayList<? super Dog> list = new ArrayList<Object>();
       list.add(new Dog());


       ArrayList<? extends Animal> list1 = new ArrayList<Horse>();
     list1.add(new Horse()); //error [Not allowed]
    }

}

import java.util.*;
class Parent
{
}
class Child extends Parent
{
}

public class Test11
{
public static void main(String [] args)
    {
        //ArrayList<Parent> lp = new ArrayList<Child>();

        ArrayList<?> lp = new ArrayList<Child>();

        ArrayList<Parent> lp1 = new ArrayList<Parent>();

        ArrayList<Child> lp2 = new ArrayList<>();

        System.out.println("Success");
    }
}

import java.util.*;
public class Test12
{
    public static void main(String[] args)
    {
        List<? extends Number> list1 = new ArrayList<Float>();

        List<? super String> list2 = new ArrayList<Object>();

        List<? super Gamma> list3 = new ArrayList<Beta>();

        List list4 = new ArrayList();

        System.out.println("yes");
    }
}

class Alpha
{
}
class Beta extends Alpha
{
}
class Gamma extends Beta
{
}

import java.util.*;

class BoundExample
{
   public static void printNumbers(List<? extends Number> list)
    {
        for (Number n : list)
        {
         System.out.println(n);
      }
   }

   public static void addNumbers(List<? super Integer> list)
    {
      list.add(100);
      list.add(200);
      list.add(300);
   }
}

public class Test13
{
   public static void main(String[] args)
    {
      // Upper Bound
      List<Integer> intList = Arrays.asList(10, 20, 30);
      List<Double> doubleList = Arrays.asList(10.5, 20.5, 30.5);

      System.out.println("Printing Integers:");
      BoundExample.printNumbers(intList);

      System.out.println("Printing Doubles:");
      BoundExample.printNumbers(doubleList);


     /* List<Character> charList = Arrays.asList('A','B','C');
      System.out.println("Printing Characters:");
      BoundExample.printNumbers(charList); */

         System.out.println("..................");

      // Lower Bound
      List<Number> numList = new ArrayList<>();
      BoundExample.addNumbers(numList);
      System.out.println("Numbers after adding: " + numList);

      List<Object> objList = new ArrayList<>();
      BoundExample.addNumbers(objList);
      System.out.println("Objects after adding: " + objList);
   }

}

class MyClass<T>     //<T extends Number>
{
    T obj;
    public MyClass(T obj)  //Student obj = new Student();
    {
        this.obj=obj;
    }

    T getObj()
    {
        return this.obj;
    }
}
public class Test14
{
    public static void main(String[] args)
    {
        Integer i=12;
        MyClass<Integer> mi = new MyClass<>(i);
        System.out.println("Integer object stored :"+mi.getObj());

        Float f=12.34f;
        MyClass<Float> mf = new MyClass<>(f);
        System.out.println("Float object stored :"+mf.getObj());

      Double d=99.34;
        MyClass<Double> md = new MyClass<>(d);
        System.out.println("Double object stored :"+md.getObj());


        MyClass<String> ms = new MyClass<>("Rahul");
        System.out.println("String object stored :"+ms.getObj());

        MyClass<Boolean> mb = new MyClass<>(false);
        System.out.println("Boolean object stored :"+mb.getObj());


        MyClass<Student> std = new MyClass<>(new Student(1,"A"));
        System.out.println("Student object stored :"+std.getObj());


    }
}

record Student(Integer id, String name)
{

}
```

E : Element Type :

```java
package com.ravi.introduction;
class Fruit
{
}
class Apple extends Fruit   //Fruit is super, Apple is sub class
{
    @Override
    public String toString()
    {
        return "Apple Fruit";
    }
}

class Basket<E>
{
    private E element;     //E is of type Fruit

    public void setElement(E element)  // Fruit element
    {
        this.element = element;
    }

    public E getElement()
    {
        return element;
    }
}

public class Test15
{
    public static void main(String[] args)
    {
        Basket<Fruit> basket = new Basket<>();
         basket.setElement(new Apple());
         Apple type = (Apple) basket.getElement();
         System.out.println(type);
    }

}
/*Generic Method
javac Test16.java
javap -c Test16.class [Disassemble the bytecode]
*/
public class Test16
{
    public static void main(String[] args)
    {
        Integer []intArr = {10,20,30,40,50};
        printArray(intArr);

        System.out.println("............");

        String []cities = {"Hyderabad", "Banglore", "Mumbai", "Kolkata"};
        printArray(cities);

    }

 //Here T describes to the compiler that method with work for any T type
    public static <T> void printArray(T[] array)
    {
        for(T element : array )
        {
            System.out.println(element);
        }
    }
}
```