

## Generics

### Introduction:

#### Case 1:

- Arrays are always type safe that is we can provide the guarantee for the type of elements present inside array.
- For example if our programming requirement is to hold String type of objects it is recommended to use String array. In the case of string array we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error.

#### Example:

```
String[] s=new String[600],  
s[0]="naresh";  
s[1]="pavan";  
s[2]= new Integer(10);//(invalid)
```

```
→ C.E → E:\SCJP>javac Test.java  
Test.java:8: incompatible types  
found   : java.lang.Integer  
required: java.lang.String
```

- That is we can always provide guarantee for the type of elements present inside array and hence arrays are safe to use with respect to type that is arrays are type safe.
- But collections are not type safe that is we can't provide any guarantee for the type of elements present inside collection.
- For example if our programming requirement is to hold only string type of objects it is never recommended to go for ArrayList.
- By mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime.

#### Example:

```
ArrayList l=new ArrayList();  
l.add("vijaya");  
l.add("bhaskara");  
l.add(new Integer(10));  
String name1=(String)l.get(0);  
String name2=(String)l.get(1);  
String name3=(String)l.get(2);(invalid)
```

```
→ R.E → Exception in thread "main" java.lang.ClassCastException:  
java.lang.Integer cannot be cast to java.lang.String
```

- Hence we can't provide guarantee for the type of elements present inside collections that is collections are not safe to use with respect to type.

**Case 2:** In the case of array at the time of retrieval it is not required to perform any type casting.

**Example:**


```
String[] s=new String[600];
s[0]="vijaya";
s[1]="bhaskara";
String name1=s[0];
```

(At the time of retrieval typecasting is not required.)

- But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

**Example:**

```
ArrayList l=new ArrayList();
l.add("vijaya");
l.add("bhaskara");
String name1= l.get(0);
```

;C.E➔ 

At the time of retrieval typecasting is Mandatory.

i.e, String name1=(String)l.get(0);

- That is in collections type casting is bigger headache.
- To overcome the above problems of collections(type-safety, type casting)sun people introduced generics concept in 1.5v hence the main objectives of generics are:

**1) To provide type safety to the collections.**

**2) To resolve type casting problems.**

- To hold only string type of objects we can create a generic version of ArrayList as follows.

```
ArrayList<String> l=new ArrayList<String>();
```

- For this ArrayList we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error that is through generics we are getting type safety.
- At the time of retrieval it is not required to perform any type casting we can assign elements directly to string type variables.

```
ArrayList<String> l=new ArrayList<String>();
l.add("A");
String name1= l.get(0);( Type casting is not required)
```

That is through generic syntax we can resolve type casting problems.

## Generics in Java

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

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

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
```

```
list.add(10);
```

```
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
```

```
list.add(10);
```

```
list.add("10");// compile-time error
```

**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

```
ArrayList<String>
```

## Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1 {
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();
        list.add("naresh");
        list.add("pavan");
        //list.add(30);//compile time error
        String s=list.get(1);//type casting is not required
        System.out.println("element is: "+s);
        Iterator<String> itr=list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

## Generic Type

Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of Java programming language. So generics also comes with its own naming conventions. Usually, type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

**E – Element**, and is mainly used by Java Collections framework.

**K – Key**, and is mainly used to represent parameter type of key of a map.

**V – Value**, and is mainly used to represent parameter type of value of a map.

**N – Number**, and is mainly used to represent numbers.

**T – Type**, and is mainly used to represent first generic type parameter.

**S – Type**, and is mainly used to represent second generic type parameter.

**U – Type**, and is mainly used to represent third generic type parameter.

**V – Type**, and is mainly used to represent fourth generic type parameter.

## Generics Classes

A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

**Note:** In Parameter type we cannot use primitives like 'int', 'char' or 'double'.

**Ex: This class is used to show the use generics class.**

```
// We use < > to specify Parameter type
class Test<T>
{
    T obj; // An object of type T is declared
    Test(T obj) { // constructor
        this.obj = obj;
    }
    public T getObject() {
        return this.obj;
    }
}

class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());
        // instance of String type
        Test<String> sObj = new Test<String>("Cvr College");
        System.out.println(sObj.getObject());
    }
}
```

**Output:15**

Cvr College

**Ex: This class is used to show the use generics class with two parameters.**

```
// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1; // An object of type T
    U obj2; // An object of type U
    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

```

class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj = new Test<String, Integer>("Cvr College", 25);
        obj.print();
    }
}

```

**Output:**

```

Cvr College
25

```

**Generic Method**

Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

**// A Generic method example**

```

class genMethod
{
    //Generic method
    <T> void genericDisplay(T element)
    {
        System.out.println(element.getClass().getName()+ " = " +element);
    }
    public static void main(String[] args)
    {
        genMethod gm=new genMethod();
        // Calling generic method with String argument
        gm.genericDisplay("Cvr College");
        // Calling generic method with Integer argument
        gm.genericDisplay(100);
        // Calling generic method with double argument
        gm.genericDisplay(25.0);
    }
}

```

**Output:**

```

java.lang.String = Cvr College
java.lang.Integer = 100
java.lang.Double = 25.0

```

## Bounded Type Parameters

Whenever you want to restrict the type parameter to subtypes of a particular class you can use the bounded type parameter. If you just specify a type (class) as bounded parameter, only sub types of that particular class are accepted by the current generic class. These are known as bounded-types in generics in Java.

### Defining bounded-types for class

You can declare a bound parameter just by extending the required class with the type-parameter, within the angular braces as –

**Syntax:** class Sample <T extends Number>

### Example

In the following Java example the generic class Sample restricts the type parameter to the sub classes of the Number classes using the bounded parameter.

```
class Test<T extends Number>{
    T a;
    Test(T a){
        this.a = a;
    }
    public void print() {
        System.out.println("value Is: "+this.a);
    }
}

public class BoundType {
    public static void main(String args[]) {
        Test<Integer> ob1 = new Test<Integer>(100);
        ob1.print();
        Test<Double> ob2 = new Test<Double>(100.11d);
        ob2.print();
        Test<Float> ob3 = new Test<Float>(1000.333f);
        ob3.print();
    }
}
```

### Output:

value is: 100

value is: 100.11

value is: 1000.333

**Note:** if you pass other types as parameters to this class (say, String for example) a compile time error will be generated.

## Bounded-types for methods

Just like with classes to define bounded-type parameters for generic methods, specify them after the extend keyword. If you pass a type which is not sub class of the specified bounded-type an error will be generated.

### Example

In the following example we are setting the Collection<String> type as upper bound to the typed-parameter i.e. this method accepts all the collection objects (of String type).

```
import java.util.*;
public class BoundTypeM {
    <T extends Collection<String>> void display(T ele){
        Iterator<String> it = ele.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
    public static void main(String args[]) {
        BTypeM b=new BTypeM();
        ArrayList<String> al = new ArrayList<String>();
        al.add("cvr");
        al.add("College");
        al.add("of");
        al.add("Engineering");
        b.display(al);
    }
}
```

**Note:** if you pass types other than collection as typed-parameter to this method, it generates a compile time error.

## Wild cards

Generics is a concept in Java where you can enable a class, interface and, method, accept all (reference) types as parameters. In other words it is the concept which enables the users to choose the reference type that a method, constructor of a class accepts, dynamically. By defining a class as generic you are making it type-safe i.e. it can act up on any datatype. To define a generic class you need to specify the type parameter you are using in the angle brackets "<>" after the class name and you can treat this as datatype of the instance variable and proceed with the code.



Instead of the typed parameter in generics (T) you can also use “?”, representing an unknown type. You can use a wild card as a –

- Type of parameter.
- Field
- Local field.

The only restriction on wilds cards is that you cannot it as a type argument of a generic method while invoking it.

Java provides 3 types of wild cards namely 1.upper-bounded, 2.lower-bounded, 3.un-bounded.

### **Upper-bounded wildcards**

Upper bounds in wild cards is similar to the bounded type in generics. Using this you can enable the usage of all the subtypes of a particular class as a typed parameter.

For example, if want to accept a Collection object as a parameter of a method with the typed parameter as a sub class of the number class, you just need to declare a wild card with the Number class as upper bound.

To create/declare an upper-bounded wildcard, you just need to specify the extends keyword after the “?” followed by the class name.

```
import java.util.*;
public class UpperBoundEx {
    public static void display(Collection<? extends Number> col){
        for (Number num: col) {
            System.out.print(num+" ");
        }
        System.out.println("");
    }
    public static void main(String args[]) {
        ArrayList<Integer> col1 = new ArrayList<Integer>();
        col1.add(24);
        col1.add(56);
        col1.add(89);
        display(col1);
        List<Double> col2 = Arrays.asList(212.1d, 33.321d, 14.49d, 42.7d, 83.4d);
        display(col2);
        HashSet<Float> col3 = new HashSet<Float>();
        col3.add(5.2f);
        col3.add(4.32f);
        display(col3);
    }
}
```

**Note:**If you pass a collection object other than type that is subclass of Number as a parameter to the display() of the above program a compile time error will be generated.

### **Lower-Bounded wildcards**

upper-bounded wildcard enables the usage of all the subtypes of a particular class as a typed parameter.

Similarly, if we use the lower-bounded wildcards you can restrict the type of the “?” to a particular type or a super type of it.

For example, if want to accept a Collection object as a parameter of a method with the typed parameter as a super class of the Integer class, you just need to declare a wildcard with the Integer class as lower bound.

To create/declare a lower-bounded wildcard, you just need to specify the super keyword after the “?” followed by the class name.

Following Java example demonstrates the creation of the Lower-bounded wildcard.

```
import java.util.*;
public class LowerBoundEx {
    void display(Collection<? super Integer> col){
        Iterator it = col.iterator();
        while (it.hasNext()) {
            System.out.print(it.next()+" ");
        }
        System.out.println("");
    }
    public static void main(String args[]) {
        LowerBoundEx lb=new LowerBoundEx();
        ArrayList<Number> col1 = new ArrayList<Number>();
        col1.add(2);
        col1.add(5);
        col1.add(8);
        col1.add(7);
        col1.add(null);
        lb.display(col1);
        List<Object> col2 = Arrays.asList(2.15f, 3.33f, 5.54f, 8.17f, 5.43f, null);
        lb.display(col2);
    }
}
```

**Note:**If you pass a collection object other of type other than Integer and its super type as a parameter to the display() of the above program a compile time error will be generated.

## Unbounded wildcards

An unbounded wildcard is the one which enables the usage of all the subtypes of an unknown type i.e. any type (Object) is accepted as typed-parameter.

For example, if want to accept an ArrayList of object type as a parameter, you just need to declare an unbounded wildcard.

To create/declare a Unbounded wildcard, you just need to specify the wild card character “?” as a typed parameter within angle brackets.

### Example for Unbounded wildcards:

```
import java.util.*;
public class wildcardEx {
    public static void display(Collection<?> col){
        System.out.println(col);
    }
    public static void main(String args[]) {
        ArrayList<Integer> col1 = new ArrayList<Integer>();
        col1.add(24);
        col1.add(56);
        col1.add(89);
        col1.add(75);
        col1.add(36);
        display(col1);
        List<Double> col2 = Arrays.asList(212.1d, 33.321d, 14.49d, 42.7d, 83.4d);
        display(col2);
        HashSet<Float> col3 = new HashSet<Float>();
        col3.add(5.2f);
        col3.add(4.32f);
        col3.add(111.22f);
        col3.add(99.25f);
        display(col3);
        List<String> col4 = Arrays.asList("cvr", "college", "of", "engineering");
        display(col4);
    }
}
```

## Inheritance & Sub Types

The inheritance is a very useful and powerful concept of object-oriented programming. In java, using the inheritance concept, we can use the existing features of one class in another class. The inheritance provides a greate advantage called code re-usability. With the help of code re-usability, the commonly used code in an application need not be written again and again.

The inheritance is the process of acquiring the properties of one class to another class.

In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.

The Parent class is the class which provides features to another class. The parent class is also known as Base class or Superclass.

The Child class is the class which receives features from another class. The child class is also known as the Derived Class or Subclass.

**Note:** In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

### **Inheritance Rules**

#### **1. A generic class can extend a non-generic class.**

```
class NonGenericClass
{
    //Non Generic Class
}
class GenericClass<T> extends NonGenericClass
{
    //Generic class extending non-generic class
}
```

#### **2. Generic class can also extend another generic class. When generic class extends another generic class, sub class should have at least same type and same number of type parameters.**

```
class GenericSuperClass<T>
{
    //Generic super class with one type parameter
}
class GenericSubClass1<T> extends GenericSuperClass<T>
{
    //sub class with same type parameter
}
class GenericSubClass2<T, V> extends GenericSuperClass<T>
{
    //sub class with two type parameters
}
class GenericSubClass3<T1, T2> extends GenericSuperClass<T>
{
    //Compile time error, sub class having different type of parameters
}
```

**3. When generic class extends another generic class, the type parameters are passed from sub class to super class same as in the case of constructor chaining where super class constructor is called by sub class constructor by passing required arguments.**

```
class GenericSuperClass<T>
{
    T t;
    public GenericSuperClass(T t)
    {
        this.t = t;
    }
}
class GenericSubClass<T> extends GenericSuperClass<T>
{
    public GenericSubClass(T t)
    {
        super(t);
    }
}
public class GenericsInJava
{
    public static void main(String[] args)
    {
        GenericSubClass<String> gen = new GenericSubClass<String>("I am string");
        System.out.println(gen.t);    //Output : I am string
    }
}
```

**Note:** In the above program 'T' in 'GenericSuperClass' will be replaced by String.

**4. A generic class can extend only one generic class and one or more generic interfaces. Then it's type parameters should be union of type parameters of generic class and generic interface(s).**

```
class GenericSuperClass<T1>
{
    //Generic class with one type parameter
}
interface GenericInterface1<T1, T2>
{
    //Generic interface with two type parameters
}
interface GenericInterface2<T2, T3>
{
    //Generic interface with two type parameters
}
class GenericClass<T1,T2, T3> extends GenericSuperClass<T1> implements
GenericInterface1<T1, T2>, GenericInterface2<T2, T3>
{
    //Class having parameters of both the interfaces and super class
}
```

**5. Non-generic class can't extend generic class except of those generic classes which have already pre defined types as their type parameters.**

```
class GenericSuperClass<T>
{
    //Generic class with one type parameter
}
class NonGenericClass extends GenericSuperClass<T>
{
    //Compile time error, non-generic class can't extend generic class
}
class A
{
    //Pre defined class
}
class GenericSuperClass1<A>
{
    //Generic class with pre defined type 'A' as type parameter
}
class NonGenericClass1 extends GenericSuperClass1<A>
{
    //No compile time error, It is legal
}
```

**6. Non-generic class can extend generic class by removing the type parameters. But, it gives a warning.**

```
class GenericClass<T>
{
    T t;
    public GenericClass(T t)
    {
        this.t = t;
    }
}
class NonGenericClass extends GenericClass    //Warning
{
    public NonGenericClass(String s)
    {
        super(s);    //Warning
    }
}
public class GenericsInJava
{
    public static void main(String[] args)
    {
        NonGenericClass nonGen = new NonGenericClass("I am String");
        System.out.println(nonGen.t);    //Output : I am String
    }
}
```

7. While extending a **generic class having bounded type parameter**, type parameter must be replaced by either upper bound or it's sub classes.

```
class GenericSuperClass<T extends Number>
{
    //Generic super class with bounded type parameter
}
class GenericSubClass1 extends GenericSuperClass<Number>
{
    //type parameter replaced by upper bound
}
class GenericSubClass2 extends GenericSuperClass<Integer>
{
    //type parameter replaced by sub class of upper bound
}
class GenericSubClass3 extends GenericSuperClass<T extends Number>
{
    //Compile time error
}
```

## Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

Using the Collections classes as an example, `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

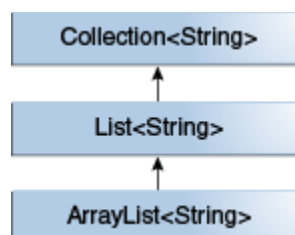


Fig:A sample Collections hierarchy

Now imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like:

```
interface PayloadList<E,P> extends List<E> {
    void setPayload(int index, P val);
    ...
}
```

The following parameterizations of PayloadList are subtypes of List<String>:

- PayloadList<String,String>
- PayloadList<String,Integer>
- PayloadList<String,Exception>

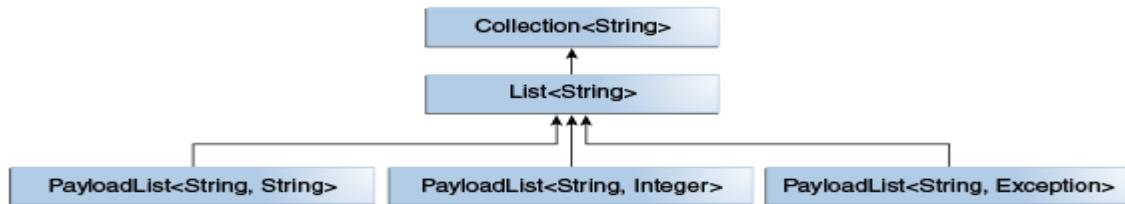


Fig:A sample PayloadList hierarchy

## Type Inference

Type inference is a feature in Java generics that allows the compiler to determine the type of a generic class or method based on the context in which it is used. In other words, it allows the programmer to omit explicit type declarations for generic types in certain situations, making code more concise and easier to read.

Java introduced type inference in Java 7, and it is implemented through a process called "diamond inference." The diamond operator (<>) can be used to declare a generic type without specifying its type parameters. The compiler infers the type parameters based on the context in which the diamond operator is used.

For example, consider the following code that declares a generic List of String objects:

```
List<String> names = new ArrayList<>();
```

Here, the type parameter String is not explicitly specified in the ArrayList constructor, but the compiler infers it from the declared type of names.

Type inference can also be used with generic methods. For example, consider the following method that takes a generic list and returns its first element:

```
public static <T> T getFirst(List<T> list) {
    if (list.isEmpty()) {
        return null;
    }
    return list.get(0);
}
```

Here, the type parameter T is inferred based on the type of the argument passed to the method.

Type inference in Java generics has some limitations. For example, it cannot be used when creating anonymous inner classes, or when working with raw types. It is also important to note that while type inference can make code more concise and readable, it can also make it less clear and more difficult to understand, especially for other programmers who may not be familiar with the context in which the type inference is used.



## Restrictions on Generics

- You can't have a static field of type
- You cannot create the static type of generic.

If static fields of type parameters were allowed, then the following code would be confusing:

```
public class MobileDevice<T>{
    private static T OS;
    // ...
}
MobileDevice<smartPhone> phone = new MobileDevice<>();
MobileDevice<pager> pager = new MobileDevice<>();
MobileDevice<tabletPC> pc = new MobileDevice<>();
```

Since the static field OS is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore, create static fields of type parameters.

- You cannot create an Instance of T
- You cannot create an instance of a type parameter T.

For instance, you cannot do the following:

```
public class GenericClass<T>
{
    public GenericClass(){
        new T();//Compile Time Error: cannot find symbol
    }
}
```

- Generics are not Compatible with Primitives in Declarations

As discussed above we cannot declare generics with primitive data types like int, float. Instead, we need to use the wrapper classes for those respective types.

```
class Student<T>{
    T age;
    Student(T age){
        this.age = age;
    }
}

public class GenericsExample {
    public static void main(String args[]) {
        Student<Float> std1 = new Student<Float>(25.5f);
        Student<String> std2 = new Student<String>("25");
        Student<int> std3 = new Student<int>(25);// Compile time error : unexpected type
    }
}
```

- You can't Create Generic Exception Class
- A generic type class cannot extend the throwable class therefore, you cannot catch or throw these objects.
- You cannot create an array of generic type objects.

```
Class Student<T>{
    T age;
    Student(T age){
        this.age = age;
    }
}

public class GenericsExample {
    public static void main(String args[]) {
        Student<Float>[] std1 = new Student<Float>[5];// Compile Time Error
    }
}
```