**Objective of generics:**

1. To provide Type Safety
2. To resolve Type Casting Problems

**Type Safety:**

Arrays are type safe i.e. we can give the guarantee for the type of elements present inside array. But Collection is not type safe. If the requirement is hold only string type of objects then we can choose String array. By mistake if we are trying to add any other type of objects we will get compile time error.

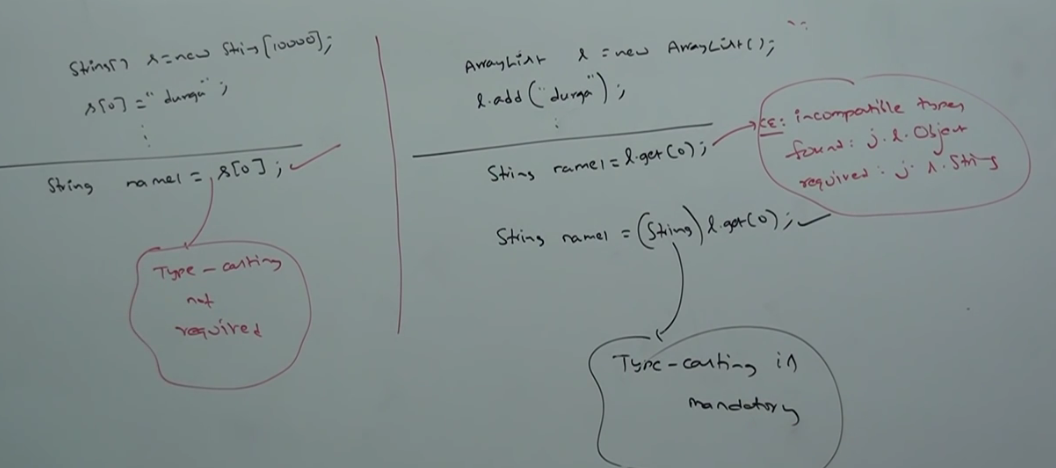
Arrays are safe to use with respect to type.

Collections are not type safe. We can’t give the guarantee for the type of elements present inside collections.

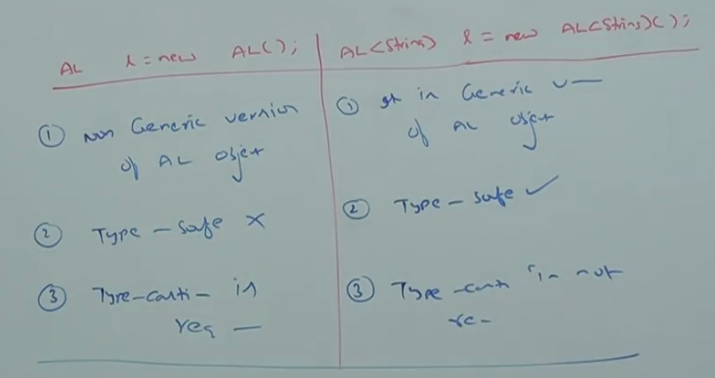
**Type Casting:**

In case of arrays at the time of retrieval, it is not required to perform type casting. Because there is a guarantee for the type of elements present inside array.

In case of collections at the time of retrieval compulsory we should perform type casting. Because there is no guarantee for the type of elements present inside collections.

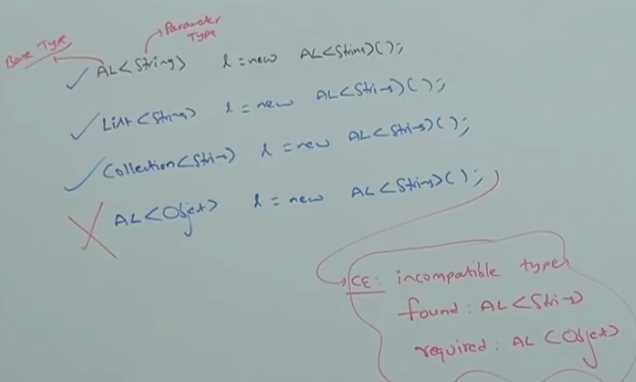


To overcome Type casting and Type Safety generics concept is used.



**Case1:**

Polymorphism (Usage of parent reference to hold child object) concept applicable only for base type not for parameter type.



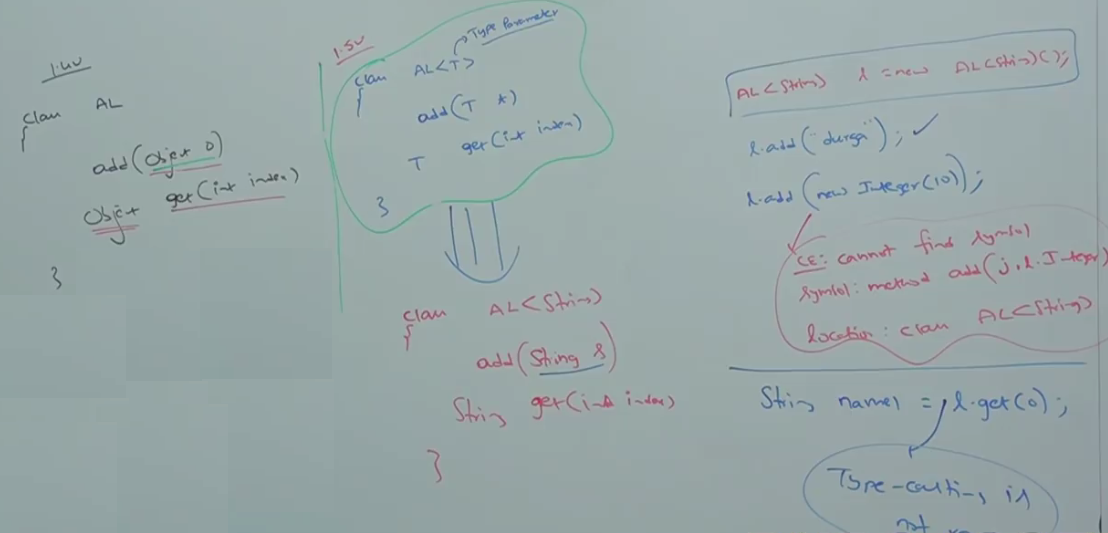
**Case2:**

**AL<int> list = new AL<int> ();** 🡪 For the type parameter we can provide, any class or interface name. But not primitive. If we are trying to provide primitive then we will get compile time error.

**Generic Classes:**

In 1.5 version a generic version of ArrayList class is declared as follows. Based on our runtime requirement T will be replaced with our provided type.

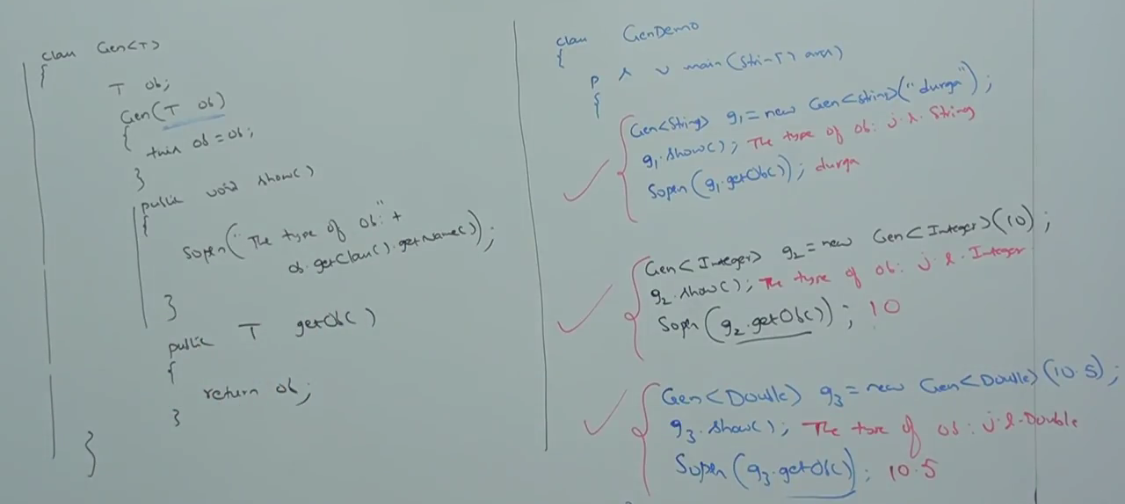
Ex: To hold only String type of objects a generic version of ArrayList object can be created as follows:



In generics we are associating a type parameter to the class. Such type of parameterized classes is nothing but Generic classes or Template classes.

We can define generic class:

Ex:



**Bounded Types:**

We can bound the type parameter for a range by using extends keyword. Such types are called: Bounded Types.

Class Test<T> {} 🡪 We can pass anything. Hence it is unbounded type.

Ex: Test<Integer> t1= new Test<Integer> ();

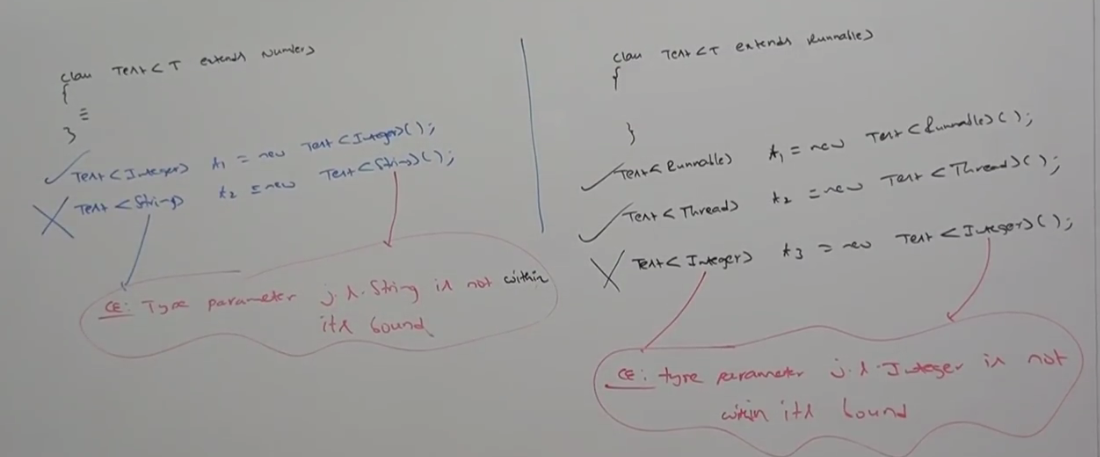
Test<String> t1= new Test<String> ();

**Syntax for Bounded Type:  
Class Test <T extends X> 🡪** X can be either class or interface.

If X is class, then we can pass X type or its child classes.

If X is an interface, then as type parameter we can pass either X type or its implementation classes.

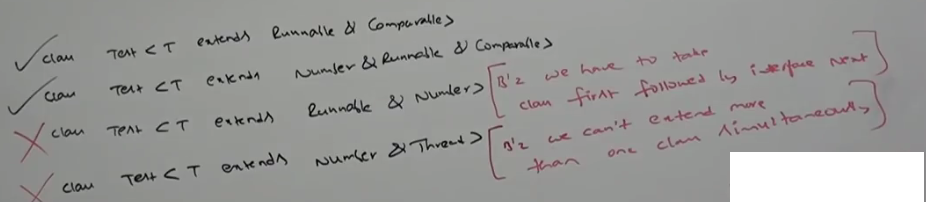
**Ex:**



Bounded types can be defined in combination as follows:

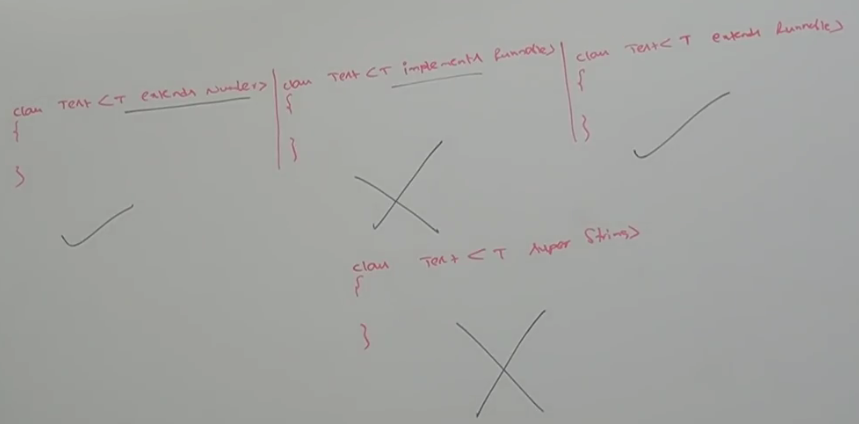
Class Test <T extends Number & Runnable >

As type parameter we can take anything which should be child class of Number and should implements Runnable Interface.

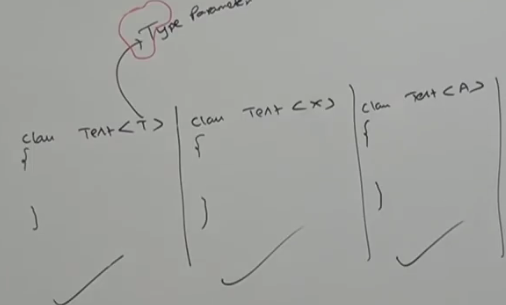


**Note\*:**

1. We can define bounded types only by using extends keyword.
2. We can’t use implements and super keywords.
3. We can replace implements keyword purpose with extends keyword.

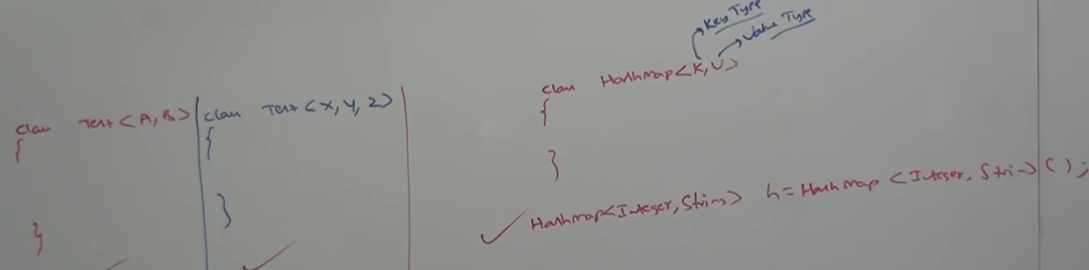


1. **As the type parameter “T”, we can take any valid java identifier. But it is good to use T.**



1. Based on the requirement we can declare any number of type parameters and all these type parameters should be separated with comma.

**Ex:**



**Generic method and Wildcard Characters(?)**

1. **M1(Al<String> al)🡪** We can call this method by passing array list of string type. Within the method we can add only string type of objects to the list. Any other type is not allowed.
2. **M1(Al<?> al)**🡪 We can call this method by passing Array List of any type. Within the method we can’t add anything to the list except null. Because we don’t know the type exactly.
3. These methods are best suitable for read only operation.
4. **M1 (Al<? extends X> al)** 🡪 X can be class or interface. If X is a class then we can call this method by passing ArrayList of either X type or its child classes.

**If X is an interface then we can call this method, by passing ArrayList of either X type or its implementation classes.**

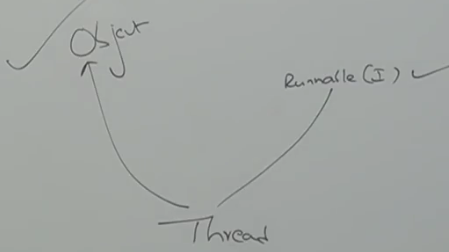
Within the method we can’t add anything to the list except null. Because we don’t know the type of X exactly. This type of method also best suitable for read-only operation.

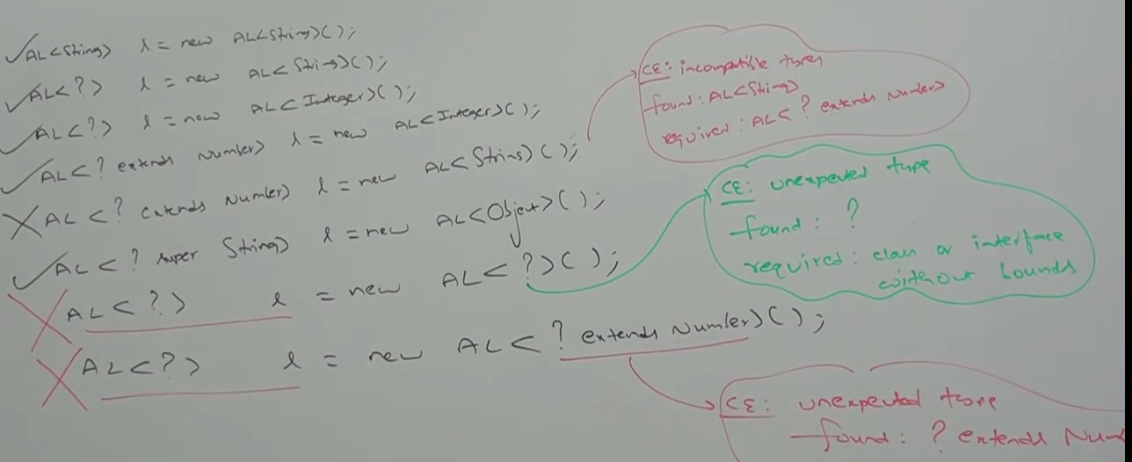
1. **M1 (Al<? super X> al)** 🡪

X can be either class or interface. If X is a class then we can call this method by passing ArrayList of either X type or its super classes.

If X is an interface then we can call this method by passing ArrayList of either X type or super

class of implementation class of X. Within the method we can add X type of object and null to the list.





We can declare type parameter either at class level or method level.

**Declaring Type parameter at class level:**

Ex: Class Test<T> {

// We can use ‘T’ within the class based on our requirement.

}

**Declaring Type parameter at method level:**

**Ex:** We must declare type parameter just before return type.

Class Test {

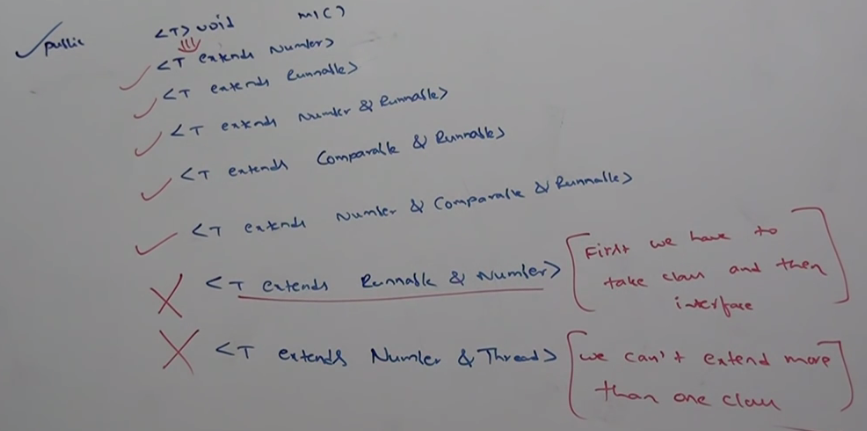
public <T>void m1 (T obj) {

//We can use ‘T’ anywhere within this method based on our requirement.

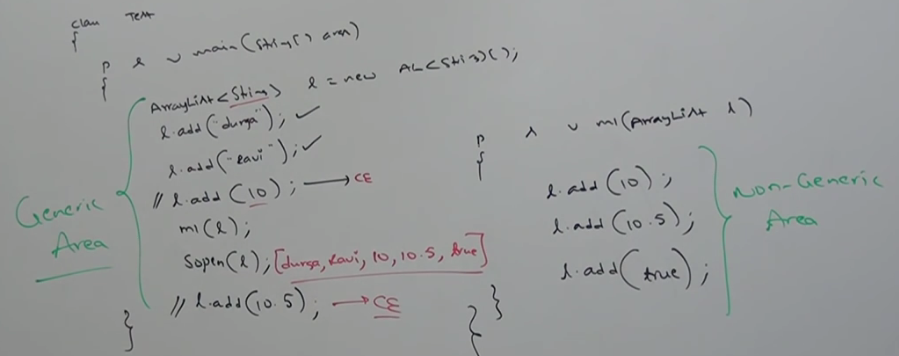
}

}

We can define bounded types even at method level also.



If we send generic object to non-generic area then it starts behaving like non- generic object. If we send non- generic object to generic area then it starts behaving like generic object i.e. the location in which object present based on that behavior will be defined.

Ex: 

The main objective of Generics is to provide type-safety and to resolve type-casting problems.

Type Safety and Type Casting both are applicable at compile time. Hence, generics concept also applicable only at compile time but not at runtime.

At the time of compilation at last step generics concept will be omitted so that at runtime this concept won’t be available to JVM.

The following declarations are equal.

AL l = new AL<String> ();

AL l = new AL<Integer> ();

AL l = new AL<Double> ();

AL l = new AL<String> ();

Ex: AL l=new AL<String> ();

l.add(10);

l.add(10.5);

l.add(true);

output= [10,10.5, true]

The following declarations are equal:

AL<String> l = new AL<String> ();

AL<String> l = new AL ();

For this ArrayList object we can add only String type of objects.

**Generics concept is applicable only at compile time:**

