**Generics**:

1. Introduction to generics
2. Introduction to generics classes
3. Bounded types in Generics
4. Introduction to Generics Method and wild card character (?)
5. Communication with non generic code
6. Conclusions

Q-: What are the purpose / need of generics concept?

A-: The main objectives of generics are:

1- To Provide type safety 2- To resolve type casting problem

**Case 1: Q-: What is type safety?**

A-: Case 1- Arrays are type safe i.e. we can give the guarantee for the type of elements present inside the arrays.

For Example: If our program requirement is to hold only string type of objects we can choose String Array. By mistake if we are trying to add any other type of objects, we will get compile type error.

|  |
| --- |
| String [] S = new String [10000];  S [0] = “Arun”;  S [1] = “Ravi”;  //s[2] = new Integer (10); Note: Here we will get compile type error (Incompatible type, found: java.lang.Integer, required: java.lang.String).  S [2] = “Shiva”; Note: This is correct because here we adding the string type object only. |

Hence string array can contain only string type of objects. Due to this we can give the guarantee for the element present inside array. Hence Arrays are safe to use w.r.t type .i.e. Arrays are type safe. So where ever a particular type is required then highly recommended for going with Arrays concept because it gives the guarantee that the element present inside the array is type safe. We cannot enter / add different type of objects inside the array.

But Collections are not type safe because we cannot give the guarantee for the type of elements present inside collections.

For example : if our program requirement is to hold only string type of object and if we choose ArrayList , by mistake if we are trying to add any other type of object we won’t get any compile time error but the program may fail at run time

Note: Now for the same requirement let me go with the collection (Array List) concept.

|  |  |
| --- | --- |
| ArrayList Al = new ArrayList ();  Al.add(“Arun”);  Al.add(“Ravi”);  Al.add(new Integer(10)); | Now we are trying to retrieve:  String name1 = (String) al. get (0);  String name 2= (String) al. get (1);  String name3 = (String) al. get (2); |

Note: Here we will get Run Time Error (Class Cast Exception) because we have added integer type in the Array List and we are trying to retrieve string.

Hence we can’t give guarantee for the type of elements present inside collections. Due to this collections are not safe to use with respect to “type”. I.e. collections are not type safe.

Note: Wherever a particular type has to represent then Arrays concept is recommended, collections are not recommended because collections are not type safe. Now the question that, what is the use of generics if we have Arrays concept already. The answer is Arrays are fixed in size. So if we want use Array then compulsory we should know the size of Arrays in advance.

But if the requirement is like that I want type safety but I don’t know the size in advance then in this case GENERICS concept is introduces (come into picture). So the main purpose of GENERICS is to provide type safety to the collections. So Collection can be made type safety by using GENERICS concept.

**Note**: In Array if we try to add incompatible data type element or object then we will get the exception on compile time that can be resolve easily but in case of ArrayList if we try to add incompatible data type element or object then data will gets added but at the time of retrieving the data it will give runtime exception that cannot be resolved

**Case 2 – Type Casting:**

In the 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**.**

|  |
| --- |
| For Example:  String [] S = new String [10000];  S [0] = “Arun”; // Here the String “Arun” will be added on the first position  Now at the time of retrieval.  String name 1= S [0]; // Here we will gets the String as “Arun” placed at first position.  **No any type casting is required at the time of retrieval of elements.** |

**But in the 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 the collection**

|  |
| --- |
| ArrayList Al = new ArrayList ();  Al.add (“Arun”); // Here the String “Arun” will be added on the first position  Now at the time of retrieval.  String name = Al.get (0); // here we will compile type exception: incompatible type, Found: java.lang.Object, Required: java.lang.String. |

So to resolve this problem we have to do type

String name = (String) Al.get (0)

Hence Type Casting is a bigger headache in Collections.

To overcome above problems of collection (Type Safety & Type Casting), sun people introduce Generic Concept in 1.5 Version.

Hence the main objectives of Generics are:

1. To Provide Type Safety 2- To Resolve Type Casting Problems

**How to get type safety in Collection: Or How to create Generics in Collection:**

To hold only String type of objects, we can create Generic Version of ArrayList object as fallows.

|  |
| --- |
| ArrayList<String> Al = new ArrayList<String> ();  Al.add (“Arun”); // Correct  Al.add (“Bunty”); // Correct  Al.add (new Integer (10)); // Not Correct because we have defined the ArrayList of String type and we are trying to add integer object , so it will throw Compile Time Exception.  Al.add (“Shiva”); |

Note: - Now at the time of element retrieval we don’t need to any type casting because we have already define the ArrayList of String type;

|  |
| --- |
| ArrayList<String> Al = new ArrayList<String> ();  Al.add (“Arun”); // Correct  Al.add (“Bunty”); // Correct  Al.add (“Shiva”); |

String name = Al.get (0); // Here we won’t get any Exception - Type Casting is not required

Hence Generic provides the guarantee for the type safety and resolve the problem of type casting of the elements inside the ArrayList. Hence through Generics we are getting type safety.

**Note**: If we define the ArrayList without generics then that case we can add any type of objects like as given below only at the time of retrieval we will have to type cast the object which is being retrieved.

|  |
| --- |
| ArrayList<String> Al = new ArrayList<String> ();  Al.add (“Arun”); // Correct  Al.add (“Bunty”); // Correct  Al.add (new Integer (10)); // Correct  Al.add (“Shiva”); // Correct |

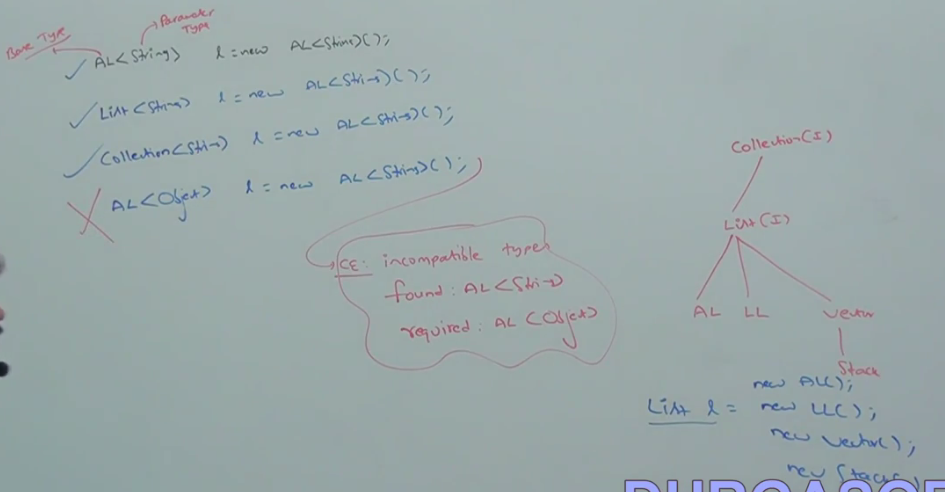
|  |  |
| --- | --- |
| **Normal ArrayList** | **Generic ArrayList** |
| ArrayList al = new ArrayList(); | ArrayList<String> al = new ArrayList<String>(); |
| 1. It is non generic version of AL object | 1. It is generic version of AL object |
| 1. Here we can add any type of object so it is not type safe. | 1. Here we cannot add an type of object so it is not type safe. We can add the object of string type only |
| 1. At the time of retrival type casting is required | 1. At the time of retrieval type casting is not required |

**Conclusion -1:**

Polymorphism concept in Generic: (uses of parent reference to hold child object is the concept of polymorphism).

Polymorphism concept applicable only for the base type but not for the parameter type

For Example:



|  |  |
| --- | --- |
| ArrayList<String> al = new ArrayList<String> (); | // Correct |
| List<String> al = new ArrayList<String> (); | // Correct |
| Collection<String> al = new ArrayList<String> (); | // Correct |
| ArrayList<Object> al = new ArrayList<String> (); | // incorrect: CE- Incompatible type  Found: ArrayList<String> Required ArrayList<object> |

In the above example as we can see that (ArrayList, List, Collection) are the Base Type and <String> is called parameter type. As in the above figure we can see that Polymorphism concept is applicable for Base type (i.e. Parent reference can hold child object) but it is not applicable for parameter type (i.e. Parent reference cannot hold child object)

Note: List and Collection is the parent class of ArrayList. Object Class is the parent class of String class.

**Conclusion 2-**

For the type parameter we can provide any class or interface name but not primitives. If we are trying to primitive then we will get compile time error

|  |  |
| --- | --- |
| Example: ArrayList<int> Al = new ArrayList<int> (); | //Incorrect It will throw and exception  CE: Unexpected Type, Found: Int, required: references: |
| ArrayList<Integer> Al = new ArrayList< Integer> (); | // Correct |

Note: Till now we came to know that Generic Provide Type Safety and resolve the problem of Type Casting. Now the next question arises how it happens internally?

**Generic Classes**

**Q: -Before Generic came into picture in Java V1.5 so until 1.4 V how the ArrayList Class was declared. ?**

**Answer**: Until 1.4 versions, a non generic version of ArrayList class is declared as fallows.

|  |
| --- |
| **Class ArrayList {**  **add (Object o );**  **Object get (int index)**  **}** |

The argument of add method is Object and hence we can add any type of objects to the ArrayList. Due to this we are missing type safety.

The return of get () method is Object, hence at the time of retrieval we have to perform type casting.

But this problem (Type Safety and Type Casting) got resolved when Java1.5 version came with Generics. So now the question arises how this problem got resolved in Java1.5 version.

So to understand this lets go to next ----Generic Classes

In 1.5 V a generic version of ArrayList class is declared as fallows.

**Class ArrayList<T> {** // Here T is a type parameter which could be any Class like String, Integer

**add (T k );**

**T get (int index)**

**}**

Based on our run time requirement T will be replaced with our provided type. For example to hold only String types of objects a generic version of ArrayList object can be created as fallows.

**ArrayList<String> Al = new ArrayList<String> ();**

For this requirement compiler considered version of ArrayList class is as fallows.

|  |
| --- |
| **Class ArrayList< String >**  **add (String k );**  **String get (int index)**  **}** |

Here every T will be replaced with String, add() method take argument of String type hence we can add only String type of Objects ,by mistake if we are trying add any other type we will get compile time error .

|  |
| --- |
| **Al. add(“Arun”); // Correct**  **Al.add(new Integer(10));** // CE: Cannot find symbol for method add(java.lang.Integer) |

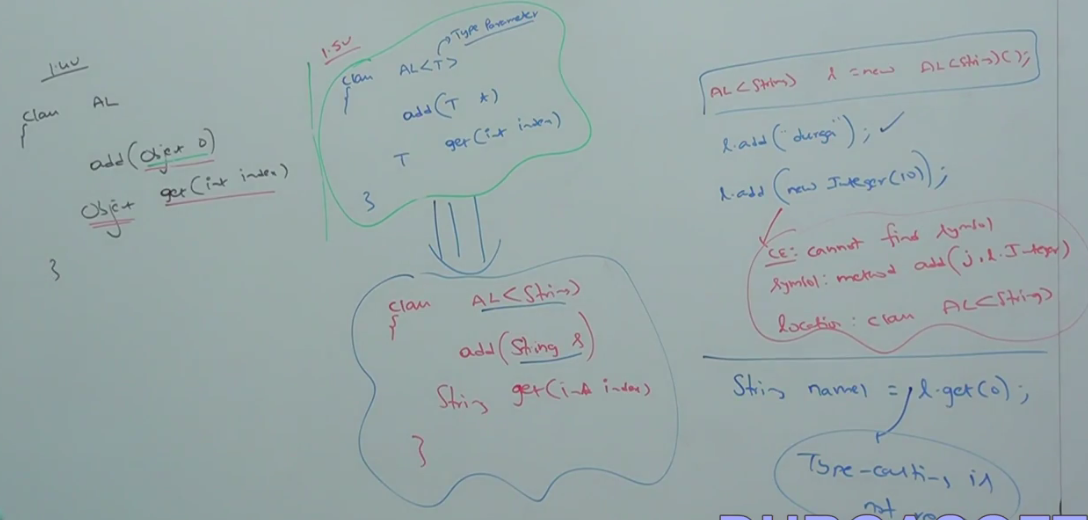
Hence through generic we are getting Type safety

The return type of get () method is String and hence at the time retrieval we are not required to perform type casting.

|  |
| --- |
| **String name1 = Al.get (0); // Type casting is not required.** |

Hence Type Safety and Type casting problem gets resolved just because of Type Parameter (T)

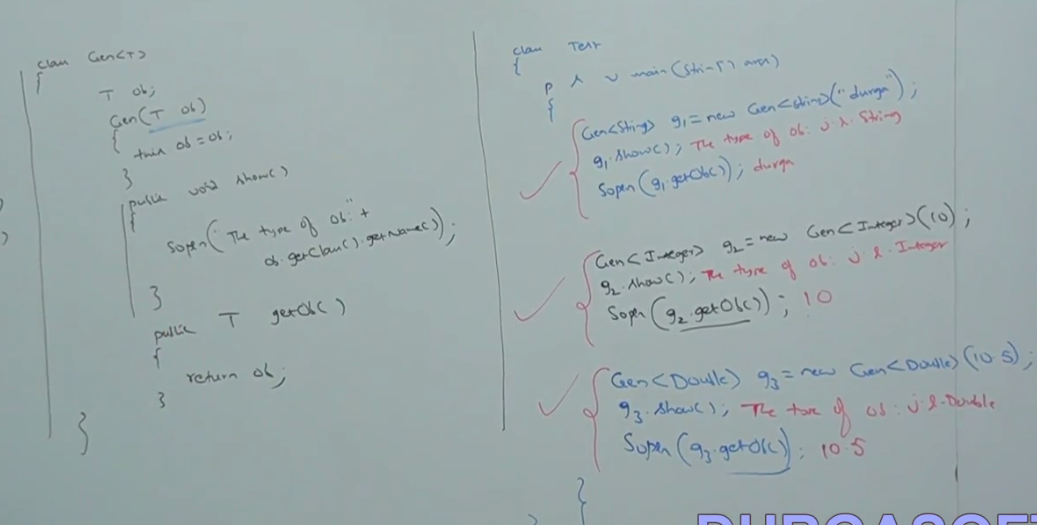
In generics we are associating a type parameter to the class, such type of parameter based classes are nothing but Generic classes or template classes. This is not new concept in java, this concept was already present in C++ template classes where based on our run time requirement we can change the type of parameter. Type Parameter can be anything , String , Integer , Student etc



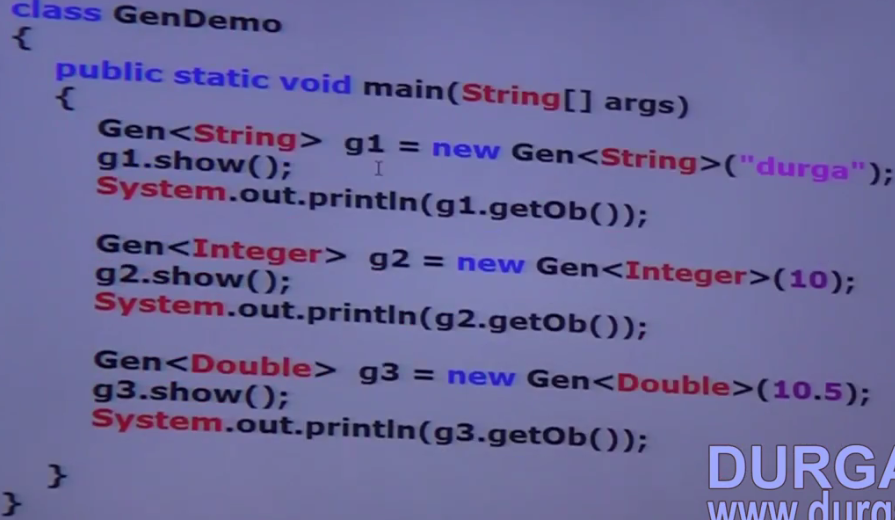
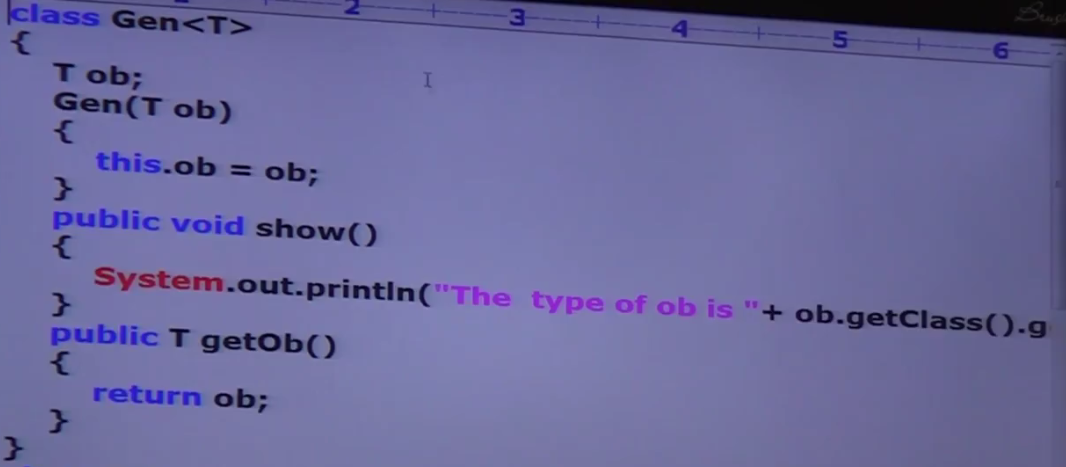
Note: Don’t feel Generic concept can be applicable only for collections. In normal java class we can also apply generic concept i.e. based on our requirement we can define our generic classes. Based on our requirement we can define our own generic classes also. Example

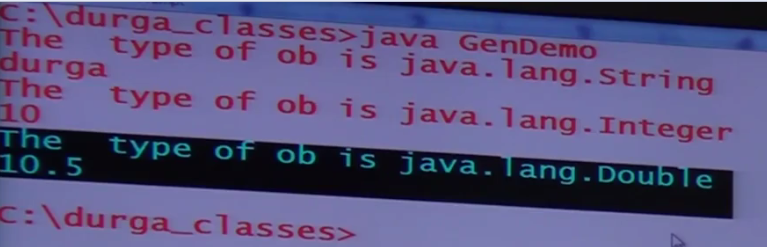
|  |
| --- |
| **Class Account<T>**  **{**  **.**  **}**  **Account<Gold> a1 = new Account<Gold> ();**  **Account<Silver> a1 = new Account< Silver > ();**  **Account<String> a1 = new Account< String > ();**  **Account<Integer> a1 = new Account< Integer > ();** |

**Example of creating our own generic class and object according to requirement**



Here in this example we are creating a generic class constructor on left side and on right side we are creating the object according to our requirements and calling the method.

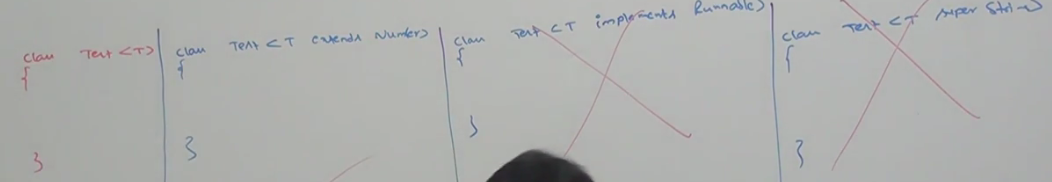




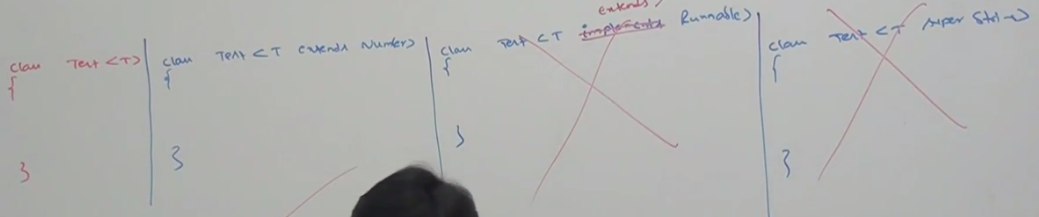
**Bounded Type in generics and how we can define bounded type:**

We can bound the type parameter (T) for a particular range by using extends Keyword. Such types are called bounded types.

|  |  |
| --- | --- |
| Class Test<T>  {  ..  }  As the type parameter ,we can pass any type and there are no restrictions and hence it is unbounded type  Example:  Test<Integer> t1 =new Test<Integer>();  Test<Integer> t2 =new Test<Integer>(); | Class Test<T **extends** Number>  {  ..  }  As the type parameter we can pass either Class Number or its child class (Integer, Float , Double )  Test<Integer> t1 =new Test<Integer>(); // Valid  Test<String> t2 =new Test< String >(); // Invalid  (it will give CE: Type Parameter java.lang.String is not within its bound) |



As we can see that **implements** Keyword is not applicable in generics but we can use the keyword **extends** and we can replace **implements** keyword with **extends** keyword in generics.

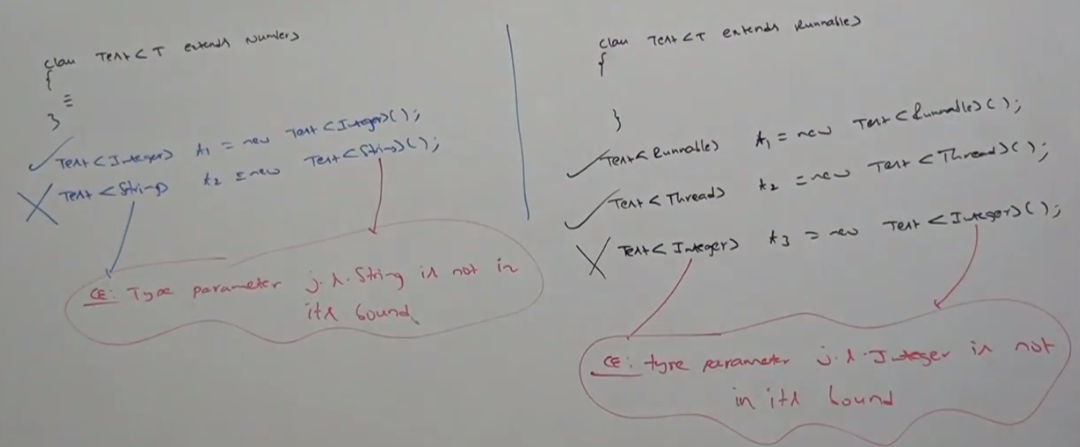


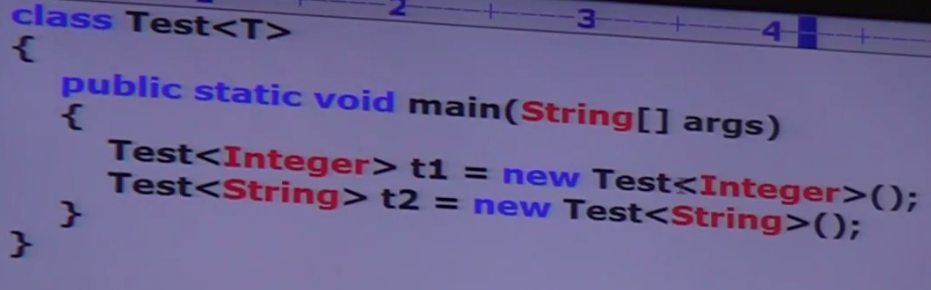
But we cannot use super keyword at all to make generic bounded

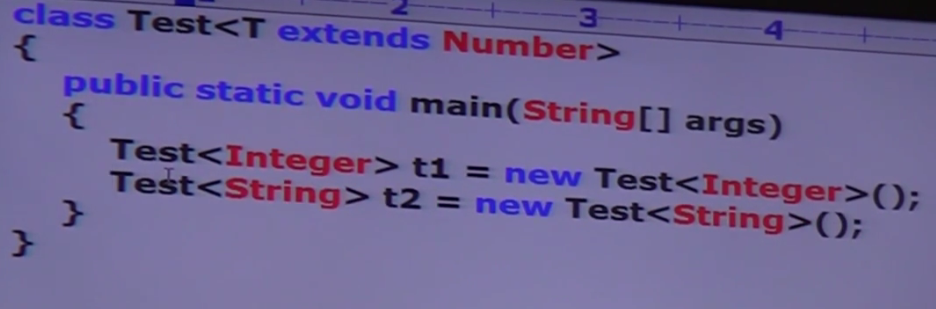
**Syntax to make generics bounded:**

|  |
| --- |
| Class Test <T extends X>  {  .  } |

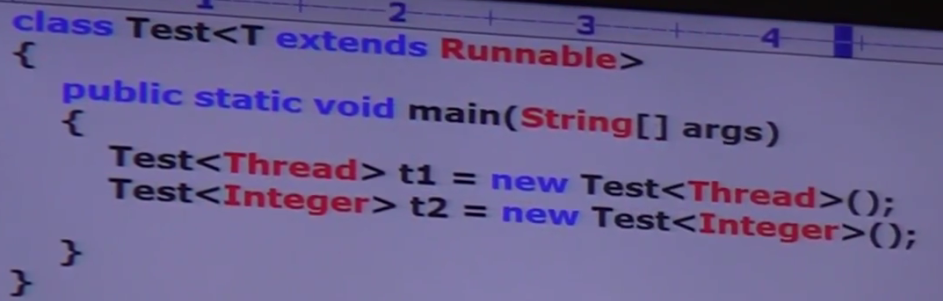
X – Can be either class or interface. If X is a class then as the type parameter we can either X type or its child classes. If X is an interface then as the type parameters either X type or its implementation classes.



Valid:

Here Test<String> t2 =new Test< String > (); // Invalid

it will give CE: Type Parameter java.lang.String is not within its bound)

 Here Test<Integer> t2 =new Test< Integer > (); // Invalid

it will give CE: Type Parameter java.lang.Integer is not within its bound)

Note: We can define bounded types even in combination also. Example

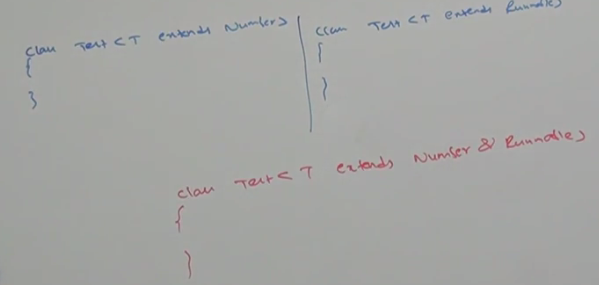
Class Test < T extends Number & Runnable>

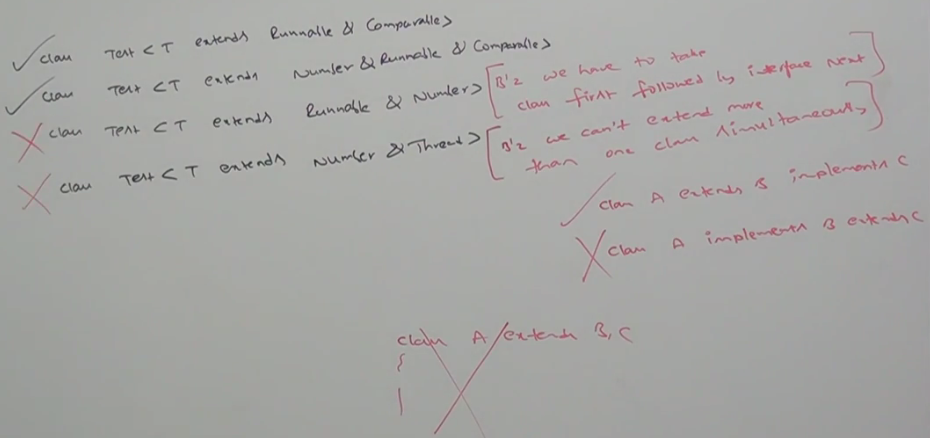
{

.

}

// as the type parameter we can take anything which should be child class of number and should implements Runnable interface



**how we can define bounded type in Generics:** 

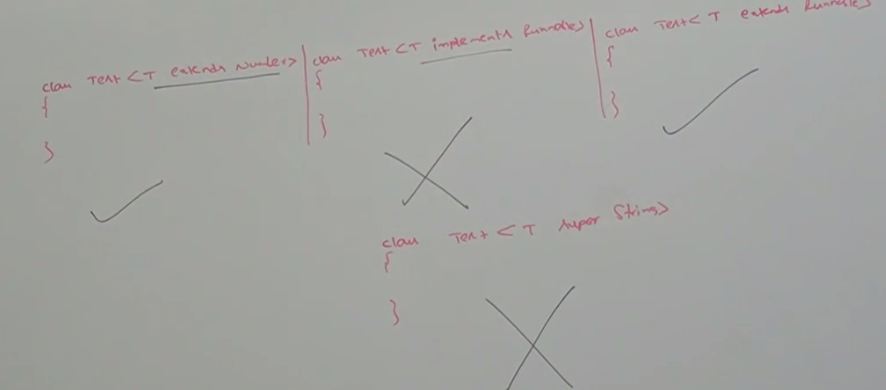
Class Test < T extends Runnable & Comparable> {} //Valid

Class Test < T extends Number & Runnable > {} //Valid

Class Test < T extends Runnable & Number> {} //Invalid because we have to take class first fallowed by Interface next i.e. first we have to extends class then We have to implements the interface

**Class Test < T extends Number & Threads> {}** //Invalid because we cannot extends more than one class simultaneously (In normal class also more than one class cannot be extends does not support multiple inheritance)

**Conclusions 1-**

We can define bounded types only by using **extends** keyword and we can’t use **implements** and **super** keywords, but we can replace implements keywords with extends keyword. 

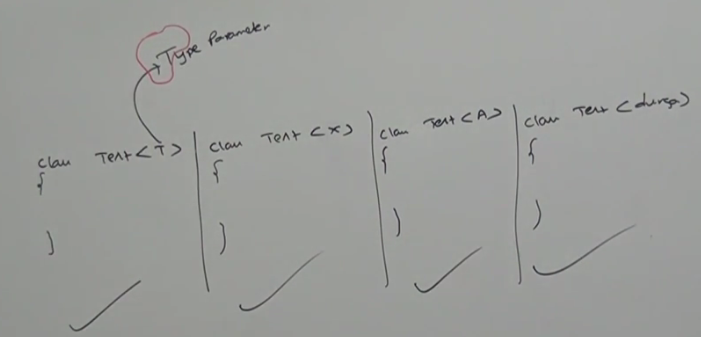
Class Test < T extends Number> {} //Valid

Class Test < T Implements Runnable> {} //InValid

Class Test < T extends Runnable> {} //Valid

Class Test < T super String> {} //InValid

**Conclusion 2-**

As the type parameter (T) , we can take any valid java identifier , but it is convention to use (T). 

Class Test < T > {} //Valid

Class Test <X > {} //Valid

Class Test < A > {} //Valid

Class Test < durga > {} //Valid

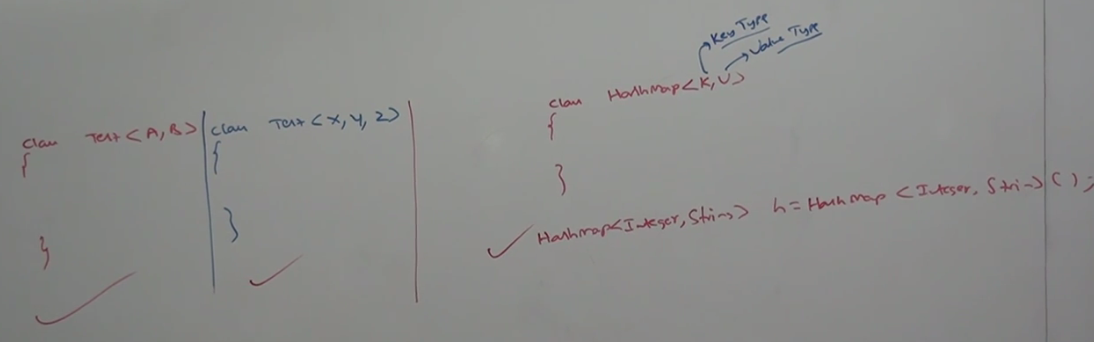
**Conclusion 3-**

**Based on our requirement we can declare any number of type parameters and all these type parameters should be separated with (,) .**

Class Test < T , W > {} //Valid

Class Test < X , Y , Z > {} //Valid

Class HashMap < K, V > {} //Valid, Here K is - key Type , V is - Value type

HashMap<Integer , String> h = new HashMap<Integer , String> ();

Session -4

**Generic Method and Wild Card Character (?):**

1. M1 (ArrayList<String > al): We can call this method by passing ArrayList of only string type. But within the method we can add only string type of objects to the list. By mistake if we are trying to add any other type then we will get Compile time error

M1 (ArrayList<String > Al)

{

Al.add(“A”); // Valid

Al.add(null); // Valid

Al.add(10); // Invalid

}

1. M1 (ArrayList<?> Al): We can call this method by passing ArrayList of any type .But within the method we can’t add anything to list except null. Because we don’t know the type exactly.

Null is allowed because it is valid value for any type .

M1 (ArrayList<?) Al):

{

Al.add(“A”); // Invalid

Al.add(null); // valid

Al.add(10); // Invalid

}

This type of method are best suitable for read only operations.

1. M1 (ArrayList<?> ) Al): X can be either class or interface

**Conclusion** We can declare type parameter <t> either at class lever or at method level.

**Declaring type parameter at class level.**

Class Test<T>

{

..

}

We can use T within class based on our requirements

**Declaring type parameter at Method level.**

We have to declare type parameter <t> just before return type

Class Test<T>

{

Public <T> void M1(T object)

{

;

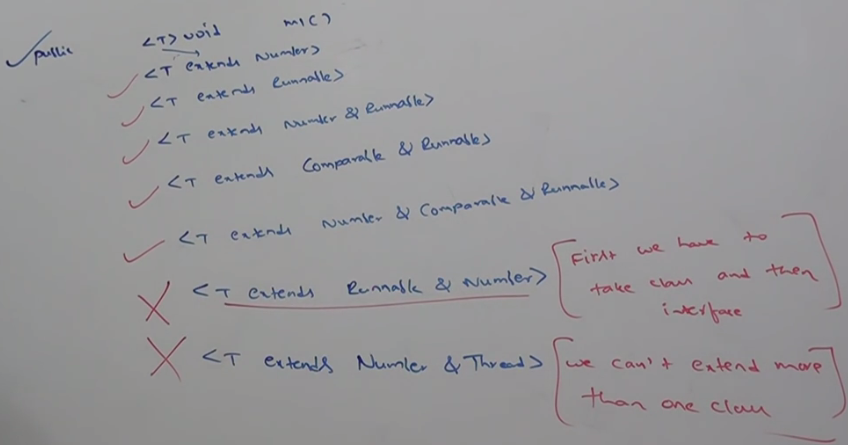
}

}

We can use T within class based on our requirements

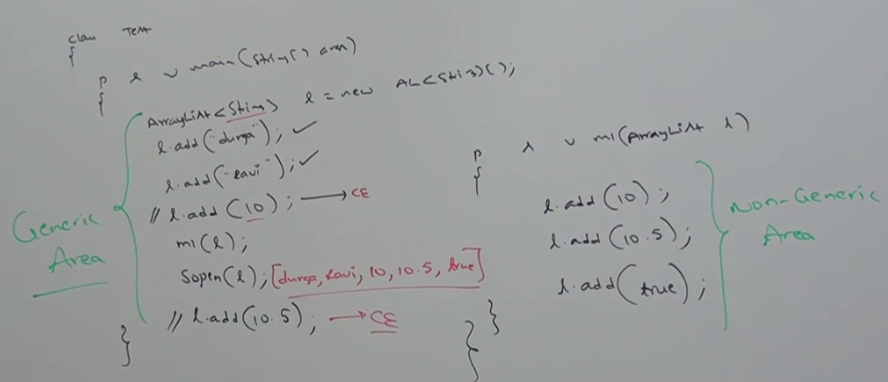
We can define bounded types even at method level also :

|  |
| --- |
| Public <T> void m1(){}  Public <T extends Number> void m1(){}  Public <T extends Runnable > void m1(){}  Public <T extends Number & Runnable> void m1(){}  Public <T extends Comprable & Runnable > void m1(){}  Public <T extends Number & Comparable & Runnable> void m1(){}  Public <T extends Runnable & Number > void m1(){} |



Day 5:

1. Communication with non generic code.

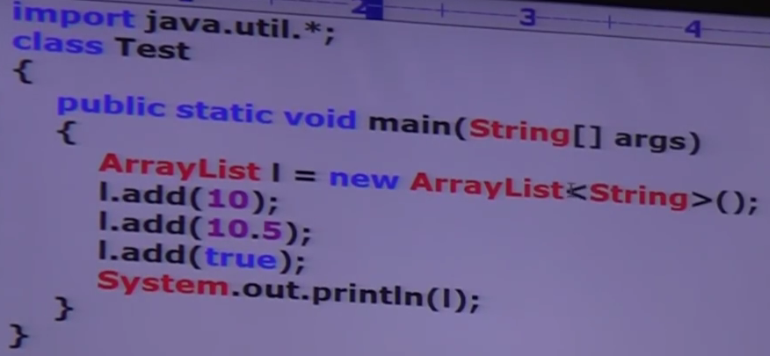
If we send generic objet to non generic area then it starts behaving link non generic object. Similarly if we send non generic object to generic area then it start behaving like generic object. i.e. the location in which object presents based on that behavior will be defined. 

Here in this example we have declared on generic Arraylist<String>

Se when we start adding string then it does not give any compile type error but when we try to add integer it gives CE. In between when we call a method which have non generic ArrayList then in this method we can add any type of object (i.e when we go from Generic Area to non generic Area it start behaving like non generic) and again in Generic Area when we try to object other than string then it gives an CE (i.e when we go from Non Generic Area to generic Area it start behaving like generic)

**Conclusions:**

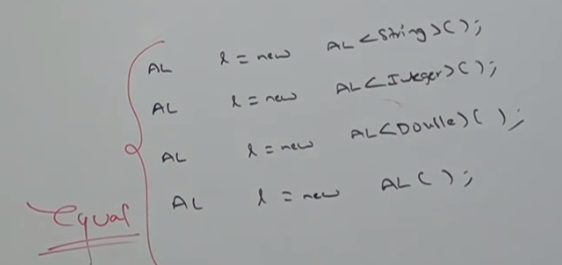
1. The main purpose of generics is to provide type safety and to resolve type casting problems.
2. Type safety and type casting both are applicable at compile time, hence generic concept also application only at compile time not at run time.
3. At the time of compilation at the last steps generic syntax will be removed and hence for JVM generic syntax won’t be available

Output: [10, 10.5, true]

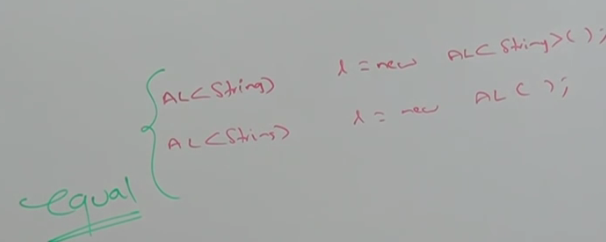
Explanation: As in the example we can see that we have taken an List at runtime like

ArrayList l = new ArrayList<String> ();

At the LHS we have not generics we have the object of normal array list so it can add any type of objects and as we know at the run time a generic behaves like non generics so at run time it will not throw any error.

Hence the following declarations are equals. 

Since generis are checked at compile time (reference type) and here at LHS we have not created any specific type so at run time it behaves the same.

The fallowing declarations are equal 

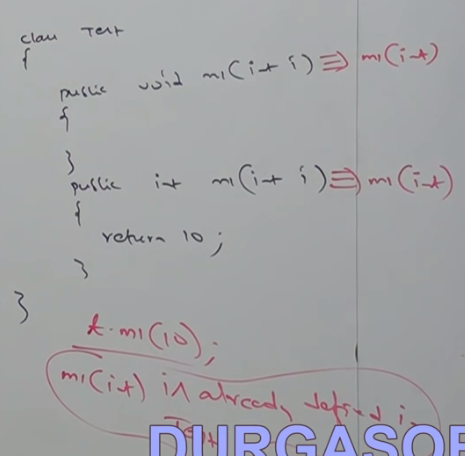
Here we can only string type object because here the ArrayList object is String type defined at LHS (reference type) and at the RHS is showing run time behavior and at run time a generics behaves as non generics that is why (new AL<String> and new AL()) are same.

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

AL <Ingeger> l = new AL ();

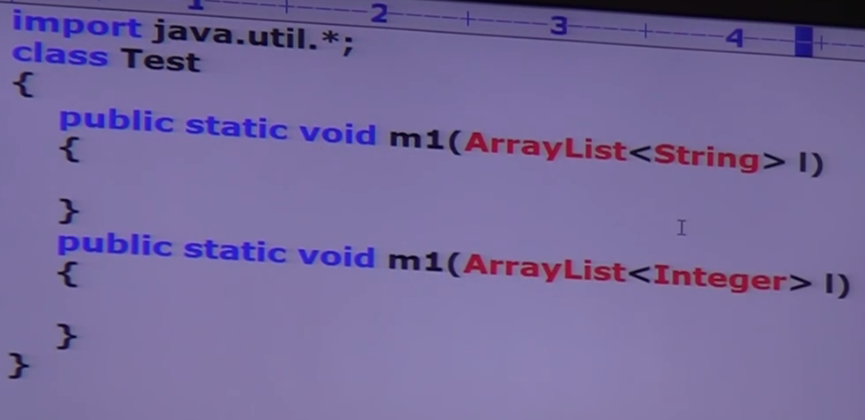
But these two are not same.

1. **Since we know two method with same signature is not allowed in java, if try then it will give compile time error that method is duplicate or method is already present with the same name**

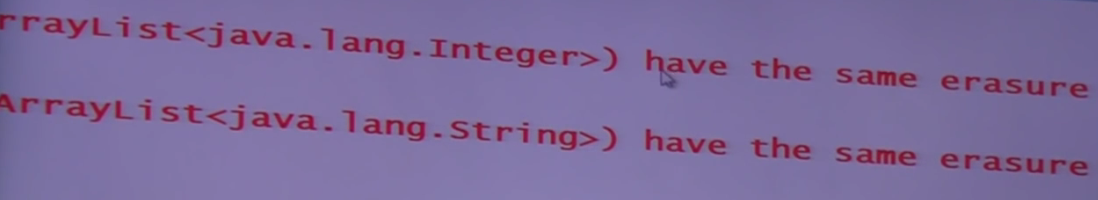
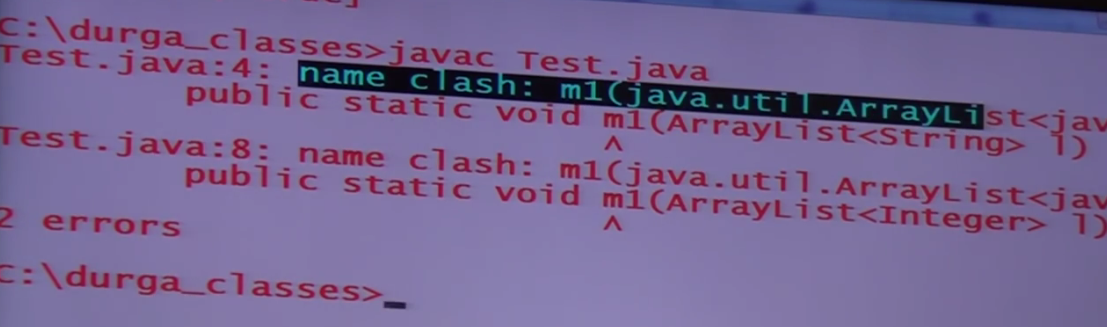


In the same way if we take the method like as give below 

Here in this example we have the same method with different signature even though it will throw CE. Because as we know in generics becomes as non generis at run time so at run time both the (ArrayList<String> l and ArrayList<Integer> l) will be converted into (ArrayList l) and hence now again the signature becomes the same and so once again we have the two method with same name and same signature that is why it will throw CE.



If we run:



Erasure means signature.