***************************************************************

# Generics:

***************************************************************

## Introduction:

- The main Objective of Generics are to provide typeSafety and to resolve type Casting Problems.
- To provide type safety and to resolve type casting.

## CASE-I:

- Arrays are TypeSafe i.e we can give the guarantee for the type of elements present inside Array.
- forex:
- If our requirement is only STring type Objects,. We can use String arrays by mistake if we are taking other types of Objects then we will get compiler error.
- String s[] = new String[1000];
- S[0] = "Chanud";
- S[1] = "10" —-> compile Error
- Arrays are safe to use with respect to type,i.e Arrays are type Safe.

ArrayList al = new ArrayList();
al.add("durga");
al.add("Chandu");
al.add(new Integer(10));

String name3 = (String) al.get(0);
Works fine but the array contains Strings and Integer also. But may problem may fail at run time.

Array is not growable and collections are growable.

Collections are not typeSafe i.e we cannot give the guarantee for the type of elements present inside the collection.
So Generics are introduced to overcome the above situation(type Safe).

## CASE-II:

EX:

```
 String s[] = new String[10];
 S[0] = “durga”;
String name1=s[0];
```

Here typecasting is not required.

EX:

```
ArrayList al = new ArrayList();
al.add(“durga”);

String name = (String)al.get(0);
```

By here At the time of retrieval we should perform typecasting, because there is no guarantee for the type of elements present inside the collection.

To overcome above problem sun people introduced generics in 1.5 v

To hold only STring TYpe of Object we can create Generic version of Arraylist Object as follows

- ArrayList<String> al = new ArrayList<String>();
- For this ArrayList we can add only String types of Objects . if you have any other type then we will get a compile error.
- EX:
- al.add(“chandu””);
- al.add(“madhu”);
- al.add(new Integer(10)); —> error

- String name1 = al.get(0);-->(no typeCasting)
- 
- From Generics we are getting Type Safety and avoid typeCasting.
-

```
ArrayList l = new ArrayList() | ArrayList<String> n= new ArrayList<String>()
                                Base-type   parameter-type
```

## Conclusion -1:

Polymorphism concept can apply for only for base type but not for Parameter type,(Usage of parent reference to hold child Object is concept of polymorphism)

```
ArrayList<String> n= new ArrayList<String>()
Basetype<Parameter-type> variable = New Object()
List<String> n= new ArrayList<String>()
Collection<String> s=new ArrayList<String>();
ArrayList<Object> o= new ArrayList<String>();  —>Error
```

## Conclusion -2:

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

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

## GENERICS CLASSES:

Until 1.4 v a non-generic version ArrayList class declared as follows.

```
Class ArrayList
{
	add(Object O)
Object get(int index);
```

}
Note: The Argument to add() is Object and we can add any type of Object to ARrayList due to this we are missing TypeSafety

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

—------------------------------

```
Class ArrayList<T>
{
	add(T O)
T get(int index);
}
ArrayList<String> n = new ArrayList();
```

But in 1.5 v a generic version released classes as follows
Based on our runtime requirement T will declare with required parameter type.

Generic in JAVA —-------- Template in c++

Based on our requirement we can define our own Generi classes also:

```
Class Account<T>
{

}
Account<Gold> n = new Account<Gold>();
Account<platinum> n = new Account<platinum>();
```

bounded types:
We can bound the type-parameter for a certain range by using extends keywords such types are called bounded types.
Ex: Class Test<T>
As the type parameter we can pass any type here are no restrictions and it is an unbounded type.

Test<Integer> t = new Test<Integer>();
Test<String> t2=new Test<String>();
Class Test<T extends Number>

Class Test<T implements Runnable> —------- X we can extend this by extends

Class Test<T super String>    - - - - X not applicable

Syntax:
Class Test<T extends X>
X can be class or interface,
If x is a class then as a type parameter we can pass either x type or its child classes., If x is interface then as type-parameter either x-type or its implementation classes.

## Unbounded:

```
Class Test<T>
{
      Public static void main(String args[])
      {
            Test<Integer> t1 = new Test<Integer>();//valid
            Test<String> t2 = new Test<STring>(); //valid
      }
}
```

## Bounded:

```
Class Test<T extends Number>
{
      Public static void main(String args[])
      {
            Test<Integer> t1 = new Test<Integer>();//valid
            Test<String> t2 = new Test<STring>(); //Invalid
      }
}
```

—----------------------------------

```
Class Test<T extends Runnable>
{
      Public static void main(String args[])
      {
            Test<Thread> t1 = new Test<Thread>();//valid
      Test<Integer> t2 = new Test<Integer>();//Invalid not in its bound
      }
}
```

—-------------------

```
Class Test<T extends Number & Runnable>
{
      Public static void main(String args[])
      {
            Test<Thread> t1 = new Test<Thread>();//valid
      Test<Integer> t2 = new Test<Integer>();      //valid
      }
}
```

As the type-parameter we can take anything which should be child class of Number and should implement runnable Interface

```
Class Test<T extends Runnable & comparable>
Class Test<T extends Number & Runnable & comparable>
```

Class Test<T extends Runnable & Number>//Invalid (class followed by Interface)
Class Test<T extends Number & Thread>//Invalid (we cant extend more than one class).

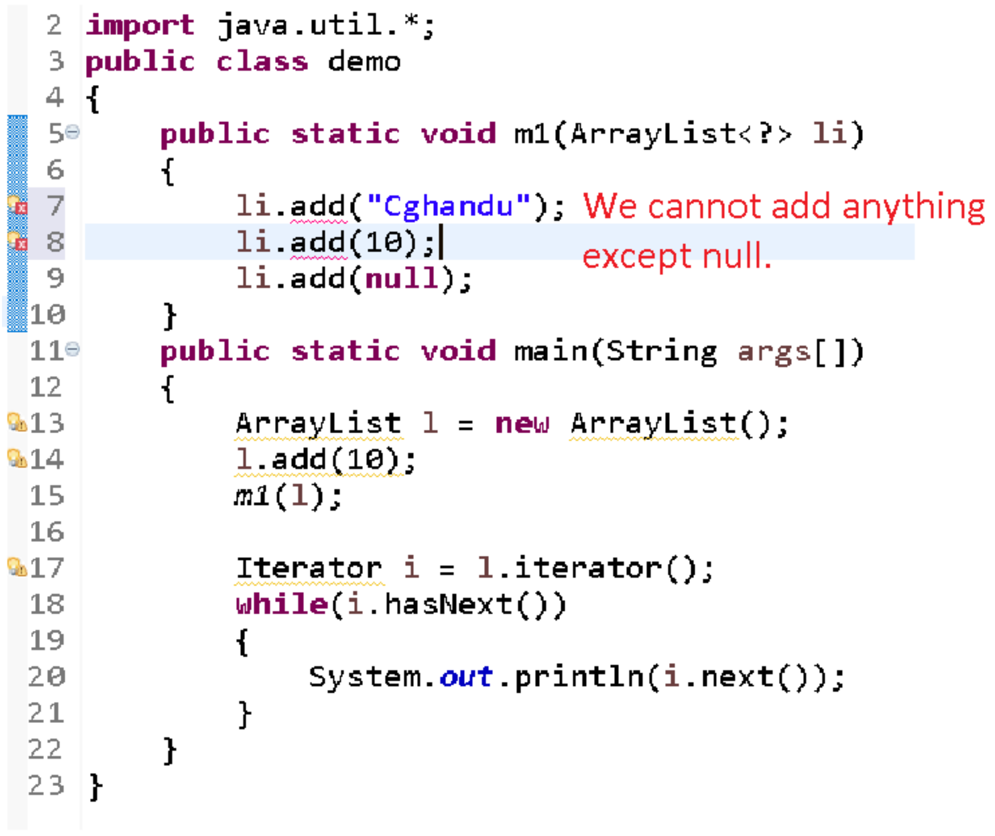
—---------------------------------------

Class Test<T> // not only T we can take any valid java Identifier, But it is convention to use "T".

Class Test<A,B> //valid
Class Test <A,B,C> //valid

Generic Methods and wildCard character?

1. m1(ArrayList<String> l);
    a. We can call this method by passing an ArrayList of Only String Type.
    b. l.add('A');
    c. l,add(null");
    d. l.add(10); //invalid
    e. Within the method we can add only String type of Object to the list, but by mistake if we are trying to add any Other type then we will get a compile time error.
2. m1(ArrayList<?> l);
    a. We can call this method by passing Arraylist of any UnKnown type(any-type)
    b. But within the method we can't add anything to list except null, because we don't known the type exactly
    c. Null is allowed because its valid value for any type,
    d. m1(ArrayList<?> l)
    e. l.add(10.5) // Invalid
    f. l.add("A") //Invalid
    g. l.add(null) //valid



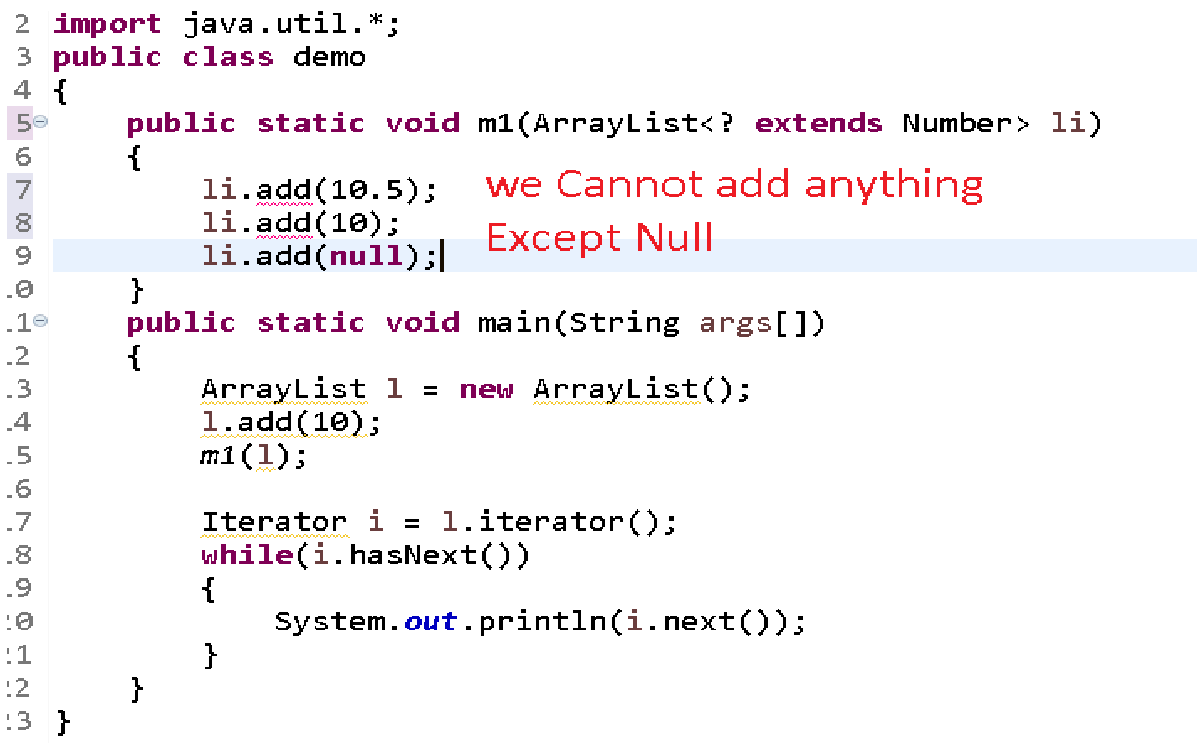
```
import java.util.*;
public class demo
{
    public static void m1(ArrayList<?> li)
    {
        li.add("Cghandu");
        li.add(10);
        li.add(null);
    }
    public static void main(String args[])
    {
        ArrayList l = new ArrayList();
        l.add(10);
        m1(l);

        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

h.

i. This type of method is best suitable for read only Operation.

3. m1(ArrayList<? Extends x> l);
    a. 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 class classes.
    b. If X is Interface then we can call this method by passing arraylist either x type are its implementation classes.
    c. But within the method we cannot add anything to the list except null because we don't know the type of x exactly, this type of method best suitable for read only Operation.



```java
import java.util.*;
public class demo
{
    public static void m1(ArrayList<? extends Number> li)
    {
        li.add(10.5);
        li.add(10);
        li.add(null);
    }
    public static void main(String args[])
    {
        ArrayList l = new ArrayList();
        l.add(10);
        m1(l);

        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

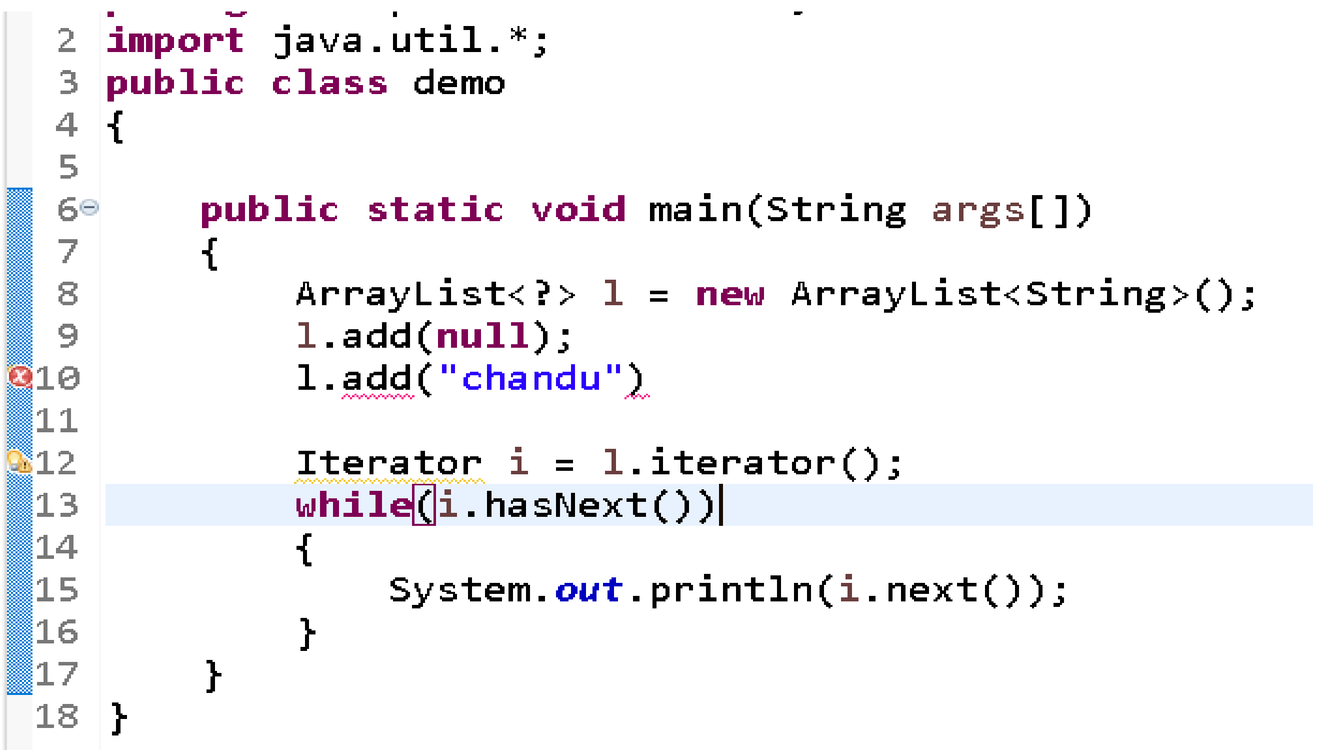
we Cannot add anything
Except Null

d.

4. m1(ArrayList<? Super x> l);
   a. X can be either class or interface
   b. If X is a class we can use this method by passing an ARrayList of x type or its SuperClasses.
   c. If X is Interface , then we can call this method passing Arraylist of x type or Super class of implementation CLass of x
   d. Object—Thread—-Ruannable
   e. We can add X-type(runnable_) or null to the list.

Valid and Invalid:

1. ArrayList<?> l= new ArrayList<String>();
2. ArrayList<?> L = new ArrayList<Integer>();
3. ArrayList <? Extends Number> l = new ArrayList<Integer>();
4. ArrayList<? Extends Number> l = new ArrayList<String>();
5. ArrayList <? Super String> l = new ArrayList<Object>();
6. ArrayList <?> l = new ArrayList<?>();
7. ArrayList <?> l = new ArrayList<? Extends Number>();//invalid required class or interface;

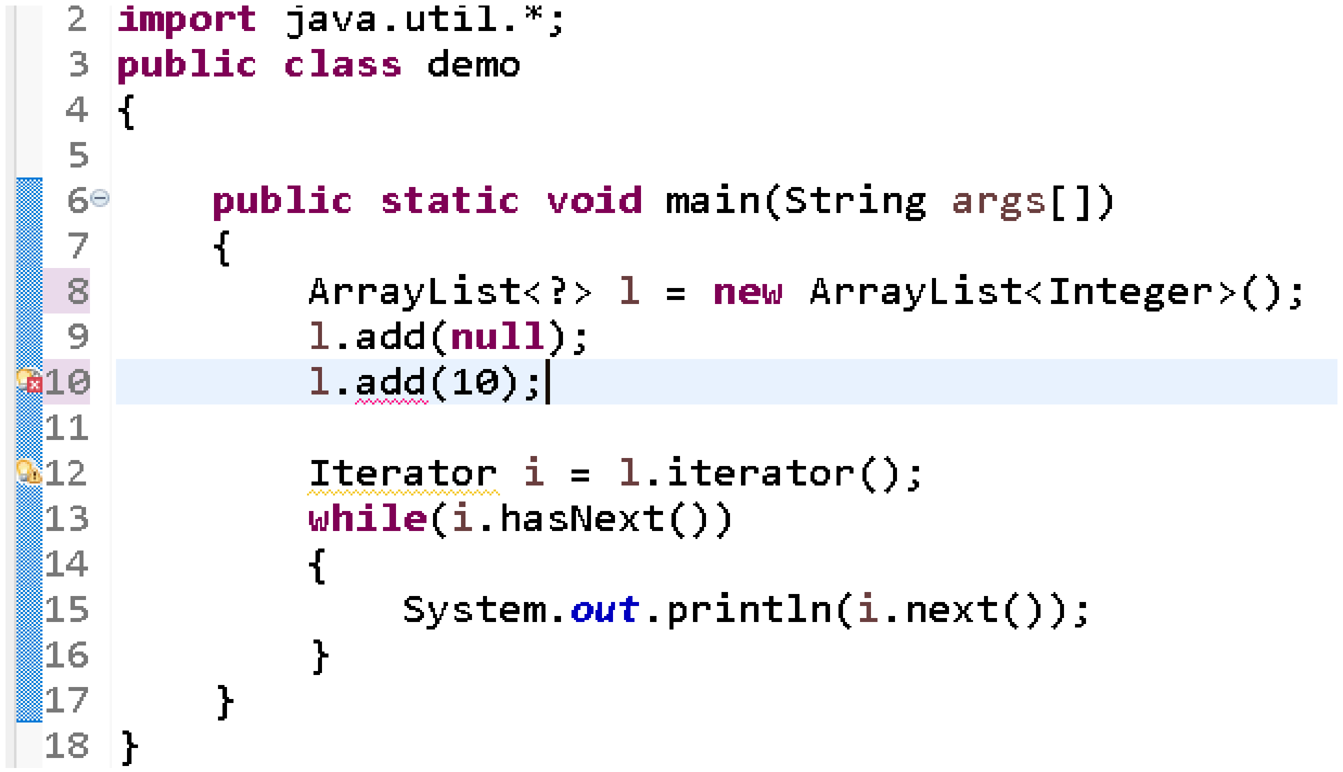


```java
import java.util.*;
public class demo
{

    public static void main(String args[])
    {
        ArrayList<?> l = new ArrayList<String>();
        l.add(null);
        l.add("chandu")

        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

1.

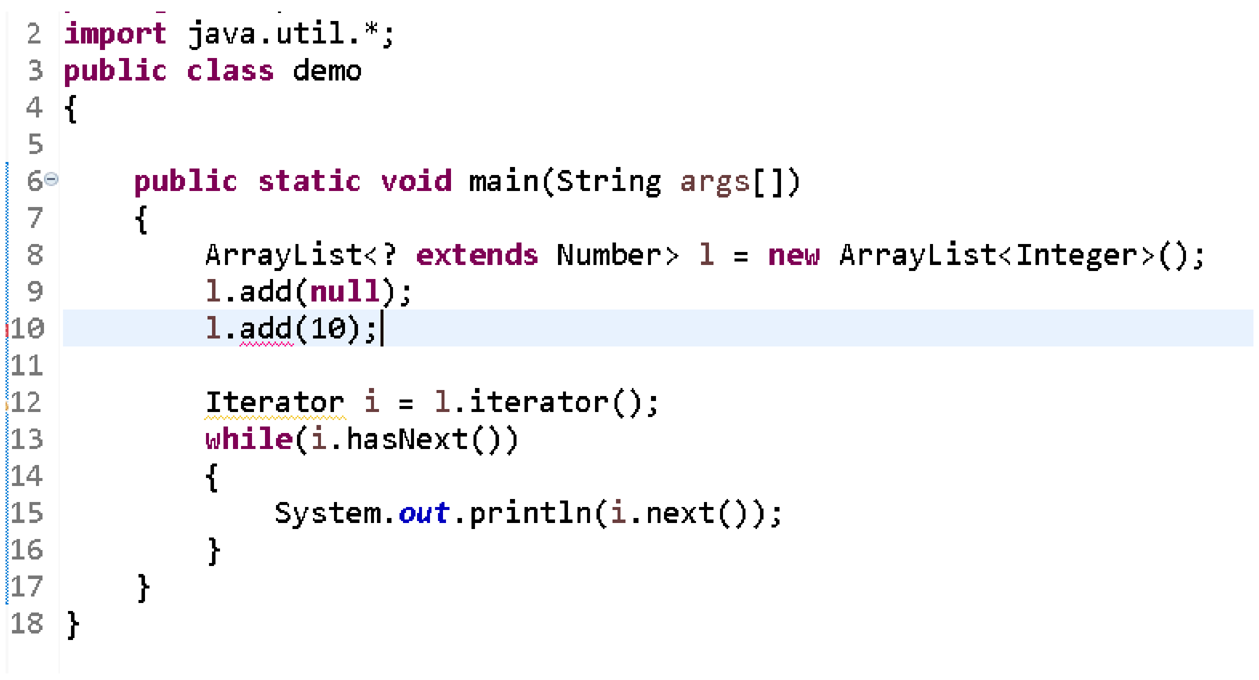


```java
import java.util.*;
public class demo
{

    public static void main(String args[])
    {
        ArrayList<?> l = new ArrayList<Integer>();
        l.add(null);
        l.add(10);

        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

2.



```
import java.util.*;
public class demo
{

    public static void main(String args[])
    {
        ArrayList<? extends Number> l = new ArrayList<Integer>();
        l.add(null);
        l.add(10);

        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

3.

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

Declaring type-parameter at class level:

Class Test<T>

We can use this T in class level based on our requirements.

Declaring type-parameter at method level

We have to declare the Type-parameter before return type

```
Class Test
{
        Public <T> void m1(T ob)
        {
        We can access T at Method level as per our requirement.
        }


}
```

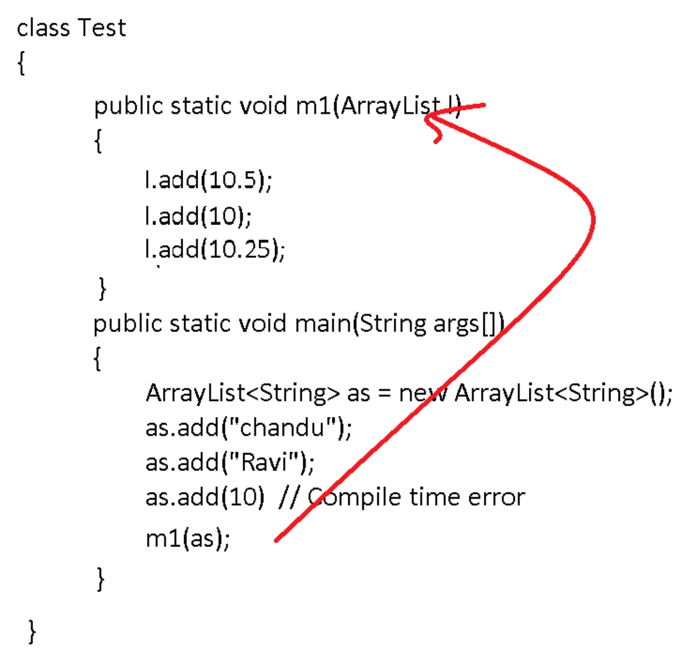
We can define Bounded ypes at method level also:

```
<T> void m1();
<T extends Number> m1();
<T extends Runnable> m1();
```

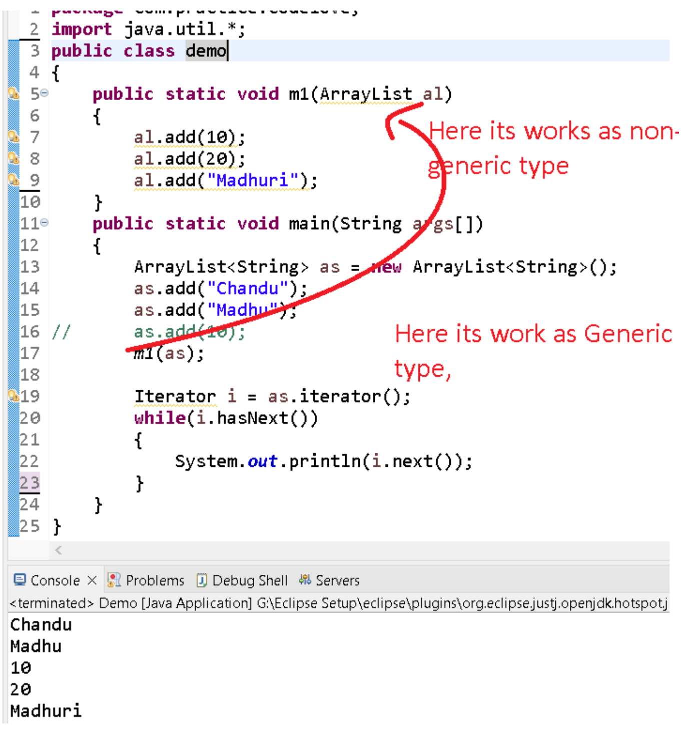
**<T extends Number & Runnable> m1();**
**<T extends Comparable & Runnable> m1();**
**<T extends Number & Comparable & Runnable> m1();**
**<T extends Runnable & Number> m1(); // Class Followed by Interface**
**<T extends Number & Thread> m1();  // No Multiple classes**

## Communication to non-generic-code:

**If we send Generic Object to non-generic area then it starts behaving live non-generic Object,If we send non-Generic Object to generic area then it starts behaving live generic Object,i.e that is the location in which present based on that Behavior will be defined.**



```
class Test
{
        public static void m1(ArrayList l)
        {
              l.add(10.5);
              l.add(10);
              l.add(10.25);
        }
        public static void main(String args[])
        {
              ArrayList<String> as = new ArrayList<String>();
              as.add("chandu");
              as.add("Ravi");
              as.add(10)  // Compile time error
              m1(as);
        }
}
```



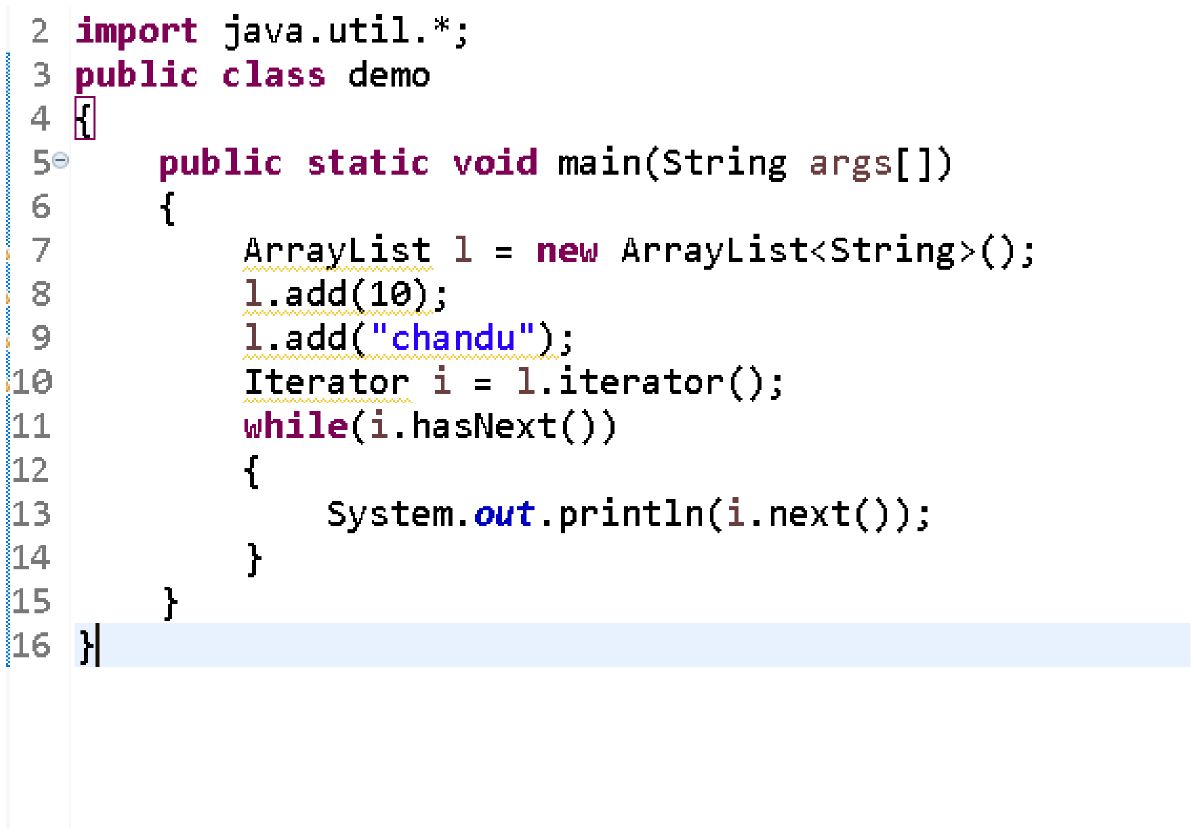
```
package com.practice.codelove;
import java.util.*;
public class demo
{
    public static void m1(ArrayList al)
    {
        al.add(10);
        al.add(20);
        al.add("Madhuri");
    }
    public static void main(String args[])
    {
        ArrayList<String> as = new ArrayList<String>();
        as.add("Chandu");
        as.add("Madhu");
//      as.add(10);
        m1(as);

        Iterator i = as.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

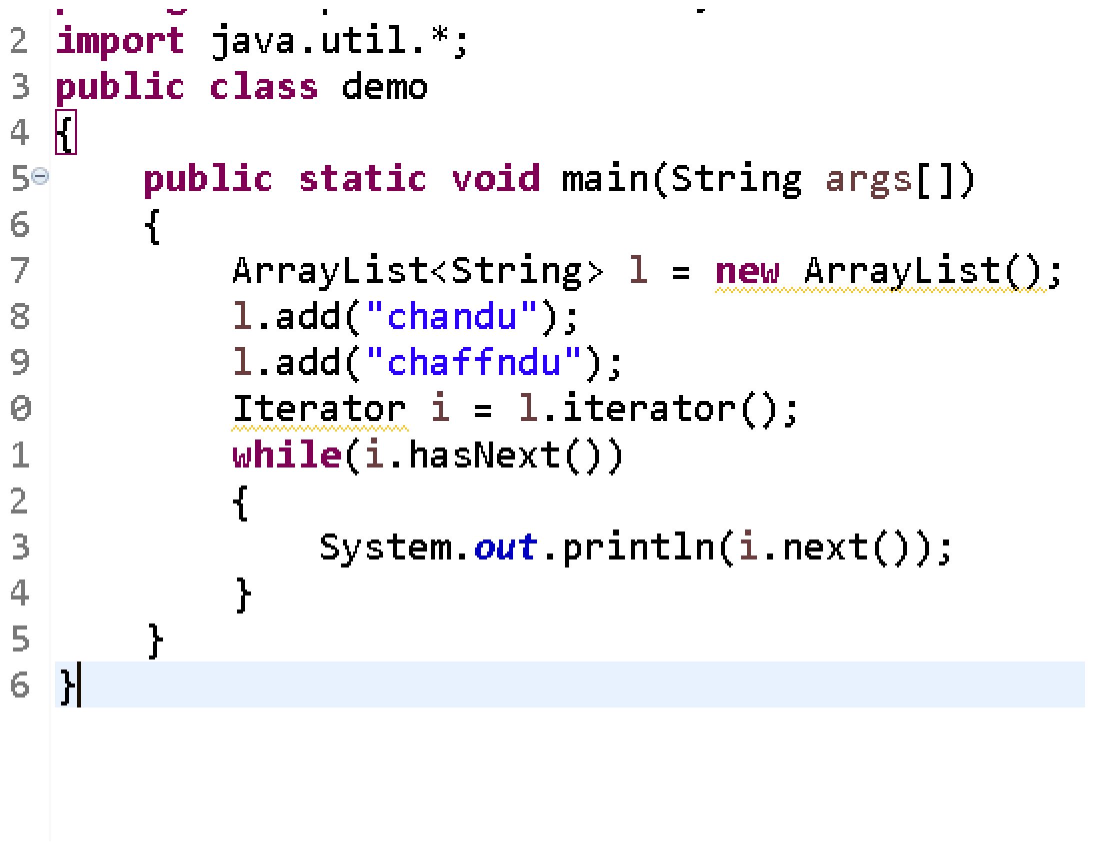
Here its works as non-generic type
Here its work as Generic type,
Console
Problems
Debug Shell
Servers

```
<terminated> Demo [Java Application] G:\Eclipse Setup\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.j
Chandu
Madhu
10
20
Madhuri
```

**The Main Purpose of Generic is to provide type safety and to resolve the type casting problem, TypeSafety and Typecasting are applicable at compile time not at runtime so Generic Concept also applicable at compile time but not at runtime. At the time of compilation at the last step Generic Synat will be removed and for JVM Generic Syntax won't be Available.**



```java
import java.util.*;
public class demo
{
    public static void main(String args[])
    {
        ArrayList l = new ArrayList<String>();
        l.add(10);
        l.add("chandu");
        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```



```java
import java.util.*;
public class demo
{
    public static void main(String args[])
    {
        ArrayList<String> l = new ArrayList();
        l.add("chandu");
        l.add("chaffndu");
        Iterator i = l.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

## Proof for (At Runtime the Generics Syntax will be removed at right Hand SIde)

Hence the following declarations are equal

```
ArrayList al = new ArrayList<String>();
ArrayList al = new ArrayList<Integer>();
ArrayList al = new ArrayList<Double>();
ArrayList al = new ArrayList();
```

Below declaration are equal:

```
ArrayList<String> al = new ArrayList<String>();
ArrayList<String> al = new ArrayList(); //
```

Compile time = Runtime

For this ArrayL:ist Object we can add only STring Type of Objects.

```
CLass Test
{
	Public static void m1(ArrayList<Integer> al)
	{
	}
	Public static void m1(ArrayList<String> al)
	{
	}

}
```

Above code give compile time error because at runtime both methods are same , so its clashes; ArrayList al.