


Today's topic

=====

1. Generics
2. Comparable vs Comparator
3. StreamAPI
4. LambdaExpression in Collection
5. JodhaApi(Date&TimeApi)

=====

```
1 public class Test {
2     public static void main(String[] args){
3         //Arrays are "TypeSafety"
4         String[] s= new String[600];
5         s[0] = "sachin";
6         s[1] = "dhoni";
7         s[2] = new Integer(10);//CE:incompatible types
8
9
10
11
12
13
14 ArrayList al=new ArrayList();
15 al.add("sachin");
16 al.add("dhoni");
17 al.add(new Integer(10));
18
19
20
21 }
22 }
```



Deff : The main objective of Generics is to provide Type-Safety and to resolve Type-Casting problems.

Case 1: Type-Safety

Arrays are always type safe that is we can give 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.

eg:

```
String name[] = new String[500];
```

```
name[0] = "Navin Reddy";
```

```
name[1] = "Haider";
```

```
name[2] = new Integer(100); // CE: incompatible types found: java.lang.Integer
```


```
required: java.lang.String
```

3

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.

```
1 public class Test {
2     public static void main(String[] args){
3
4         // I need to hold only String type of data
5         // which one to choose Arrays or Generics
6
7         // Arrays ==> directly deals the data at memory level
8         // Generics ==> deals the data through API code
9
10
11
12 }
13 }
14 }
```



```
...  
String name1 = (String)a1.get(0);  
String name2 = (String)a1.get(1);  
String name3 = (String)a1.get(2); //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: Type-Casting

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

eg::

```
String name[] = new String[500];  
name[0] = "Navin Reddy";  
name[1] = "Haider";  
...  
...  
String data = name[0]; //here type casting is not required.
```



1

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

eg::

```
...  
String data =name[0]; //here type casting 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.

eg::

```
ArrayList al =new ArrayList();  
al.add("NavinReddy");  
al.add("Haider");  
String name1= al.get(0); //CE: incompatible types : found : java.lang.Object  
                        required: java.lang.String
```


String name1=(String) al.get(0); //At the time of retrieval type casting is mandatory

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.

```
1 import java.util.*;
2
3 public class Test {
4     public static void main(String[] args){
5
6
7
8         //TypeSafetly becoz of non-generic version code
9         //ArrayList<String> a1 = new ArrayList<String>();
10        a1.add("sachin");
11        a1.add("dhoni");
12
13        //TypeCasting is required
14        String name1 = a1.get(0);
15        String name2 = a1.get(0);
16    }
17 }
```



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

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

```
al.add("NavinReddy");
```

```
al.add(10); // CE: can't find symbol
```

```
symbol: method add(int)
```

```
location : class java.util.ArrayList<java.lang.String>
```

```
al.add(10)
```

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.

eg:

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

```
al.add("NavinReddy");
```

```
....
```

```
....
```

```
String name = al.get(0); // type casting is not required as it is an TypeSafe
```

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

Ln 36, Col 69

22°C
Brisby, cloudy

Q Search




100%

Windows (CTRL)

UTF-8

20:37
16-12-2022

```
1 import java.util.*;
2
3 public class Test {
4     public static void main(String[] args){
5
6         //<String> ==> TypeParameter
7         //List ==> BaseParameter
8
9         List<String> l1 = new ArrayList<String>();//valid
10
11         Collection<String> l2 = new ArrayList<String>();//valid
12
13         ArrayList<Object> l3 = new ArrayList<String>();//invalid
14
15     }
16
17 }
18
19
```



Generics part1

```
19_12_2022_Generics_Class_Notes - Notepad
File Edit View
class ArrayList
{
    add(Object o);
    Object get(int index);
}

Generics(1.5V)
=====
|=> TypeParameter
class ArrayList<T>
{
    add(T t);          =====> add(String t)
    T get(int index);  =====> String get(int index)
}

ArrayList<String> al = new ArrayList<String>();
al.add("sachin");
al.add(new Integer(10));
```

