- If we are depending on natural sorting order,then those keys should be homogenous and it should be Comparable otherwise ClassCastException.
- If we are working on customisation through Comparator,then those keys can be heterogeneous and it can be NonComparable.
- No restrictions on values, it can be heterogeneous or NonComparable also.
- If we try to add null Entry into TreeMap, it would result in "NullPointerException".

Hashtable:

- The Underlying Data Structure for Hashtable is Hashtable Only.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.
- Heterogeneous Objects are Allowed for Both Keys and Values.
- null Insertion is Not Possible for Both Key and Values. Otherwise we will get Runtime Exception Saying NullPointerException.
- It implements Serializable and Cloneable,but not RandomAccess.
- Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread Safe,so best suited when we work with Search Operation.

Need of Generics and Basics of Generics
Generics

The purpose of Generics is

1. To provide TypeSafety.
2. To resolve TypeCasting problems.

Case1:
TypeSafety

- A guarantee can be provided based on the type of elements.
- If our programming requirement is to hold only String type of Objects,we can choose String Array.
- By mistake if we are trying to add any another type of Objects, we will get "CompileTimeError".

eg#1.

```
String[] s=new String[10000];
     s[0] = "dhoni";
     s[1] = "sachin";
     s[2] = new Integer(10); //CE:
incompatible types found :
java.lang.Integer

                  required:
java.lang.String
```

W.r.t Arrays we can guarantee that what type of elements is present inside Array, hence Arrays are safe to use w.r.t type,so Arrays as TypeSafety.

eg#2.

```
ArrayList l=new ArrayList();
     l.add("dhoni");
     l.add("sachin");
     l.add(new Integer(10));
        ...
        ....
     String s1=(String)l.get(0);
     String s2=(String)l.get(1);
     String s3=(String)l.get(2); //
RE: ClassCastException
```

For Collections, we can't guarantee the type of elements present inside Collection.
If our program requirement is to hold only String type of Objects then if we choose ArrayList by mistake if we are trying to add any other type of Object,we won't get any CompileTimeError,but the program may fail at runtime.

Note: Arrays are TypeSafe,whereas Collections are not TypeSafe.
Arrays provide guarantee for the type of elements we hold, whereas Collections won't provide guarantee for the type of elements.

Need of Generics

1. Arrays to use we need to know the size from the beginning,but if we don't the size and still If we want to provide type casting we need to use "Collections along with Generics".

Case2:
Type casting

eg#1

```
String s[]=new String[3];
   s[0] = "sachin";
 String name=s[0];//
```

Typecasting not required as we it holds only String elements only.

eg#2

```
ArrayList l=new ArrayList();
   l.add("sachin");
 String name=l.get(0);//
```

CE: found : java.lang.Object required: java.lang.String

```
String name=(String)l.get(0);
```
        |=>TypeCasting is compulsory when we work with Collections.

To Overcome the above mentioned problems of Collections we need to go for Generics in 1.5V, which provides TypeSafety and to Resolve TypeCasting problems**.**

How TypeSafety is provided in Generics and how it resolves the problems of TypeCasting?

eg#1
ArrayList

```
al=new ArrayList();//
```

Non-Generic ArrayList which holds any type of elements.

```
al.add("sachin");
 al.add("dhoni");
 al.add(new Integer(10));
            al.add("yuvi");
```

eg#2.

ArrayList

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

Generic ArrayList which holds only String.

```
al.add("sachin");
 al.add("dhoni");
 al.add(new Integer(10));//CE
            al.add("yuvi");
```

Note: Through Generics TypeSafety is provided.

ArrayList

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

Generic ArrayList which holds only String.

```
al.add("sachin");
 al.add("dhoni");
            al.add("yuvi");
 String name=al.get(0); //
```

TypeCasting is not required

At the time of retrieval, we are not required to perform TypeCasting.
Note: Through Generics TypeCasting problem is solved.

Difference b/w
ArrayList l=new ArrayList();

- It is a non generic version of ArrayList Object.
- It won't provide TypeSafety as we can add any elements into the ArrayList.
- TypeCasting is required when we retrieve elements.

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

- It is a generic version of ArrayList Object.
- It provides TypeSafety as when we can add only String type of Objects.
- TypeCasting is not required when we retrieve elements.

Conclusion - 1

```
⟾ parameter type
      ArrayList<String> al =new ArrayList<String>();
          ⟾ BaseType

      List<String> al =new ArrayList<String>();
      Collection<String> al =new ArrayList<String>();

      ArrayList<Object> al=new ArrayList<String>();//CE:incompatible type:
        found ArrayList<String>
        required ArrayList<Object>
```

Polymorphism is applicable only for the BaseType,but not for the Parameter type.
Polymorphism=> usage of parent reference to hold Child object is the concept of "Polymorphism".

Conclusion - 2

Collection concept is applicable only for Object,it is not applicable for primitive types.
So parameter type should be always be class/interface/enum,if we take primitive it would raise in "CompileTimeError".

eg#1
ArrayList

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

CE: unexpected type: found int required reference

Generic classes

until 1.4 version, nongeneric version of ArrayList class is declared as follows