

**Core Java**

**Session3 By Saurabh Sharma**

# Agenda

**Today we will cover the following two modules:**

- ❑ Generics And Collections.
- ❑ Wrapper classes and Autoboxing, Enum, Var-Args & Scanner class.

## Session : Objectives

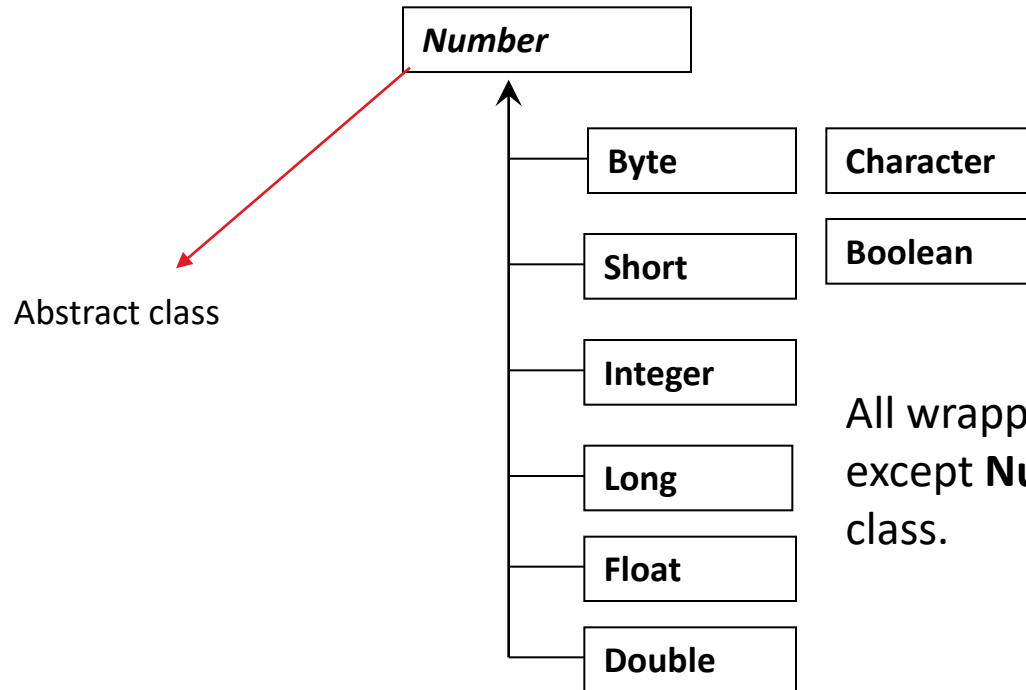
At the end of Session 3, you should be able to:

- ❑ Explain Wrapper class, enum, var-args and scanner class.
- ❑ Define collections and generics.

# Wrapper classes

- ❑ Java uses primitive types, such as int, char, double to hold the basic data types supported by the language.
- ❑ Sometimes it is required to create an object representation of these primitive types.
- ❑ These are collection classes that deal only with such objects. One needs to wrap the primitive type in a class.
- ❑ To satisfy this need, java provides classes that correspond to each of the primitive types. Basically, these classes encapsulate, or wrap, the primitive types within a class.
- ❑ Thus, they are commonly referred to as type wrapper. Type wrapper are classes that encapsulate a primitive type within an object.
- ❑ The wrapper types are Byte, Short, Integer, Long, Character, Boolean, Double, Float.

# Primitive Wrappers



All wrapper classes are **final** classes except **Number** which is **abstract** class.

## Primitive Wrappers(Constructors and common methods)

- ❑ General form of constructor for all wrapper classes:
  - ❖ Constructor with its corresponding primitive type.
  - ❖ Constructor with a **String** type. Excludes: **Character**.

Example:

**Integer(String)** and **Integer(int)**

**Double(String)** and **Double(double)**

- ❑ **Float** has an extra constructor that takes **double**.

**Float(double)**

- ❑ Also note that-

~~**Byte b = new Byte(1);**~~ // compilation error //Line1

- ❑ Also all the classes have implemented **equals()**, **toString()** and **compareTo()** methods.

# Autoboxing

- ❑ Autoboxing refers to the conversion of wrapper class type to its primitive type (and vice versa) automatically. This was included in JSE 5.0.
- ❑ Boxing is conversion of primitive to its wrapper type:
  - ❖ **Integer ii=10; ii++;**
  - ❖ **Boolean bb=true; if(bb){}**
  - ❖ **Long ll=34L;** but **Long ll=34** leads to error
  - ❖ **Byte bt=34;** but **Byte bt= 1000;** leads to error
  - ❖ **Float fl= 3.14f;** but **Float fl= 3.14;** leads to error
  - ❖ **Double dl= 3.14;**
  - ❖ **Character c='a';**
- ❑ You will find that literal conversions here is very similar to what we had for primitives.
- ❑ Unboxing is the reverse. That is a wrapper objects automatically convertible to its primitive.
  - ❖ **int i=ii; boolean b=bb; long l=ll;**
- ❑ Boxing and unboxing may hit performance. So be careful where you use it.

## Autoboxing continued

- ❑ When compiler allows Integer ii=10, what it does is it converts the code as-  
Integer ii = new Integer(10); ?
- ❑ Java distinguishes between primitive types and Objects:
  - ❖ Primitive types, i.e., int, double, are compact, support arithmetic operators.
  - ❖ Object classes (Integer, Double) have more methods: Integer.toString()
  - ❖ You need a “wrapper” to use Object methods:  
Integer ii = new Integer( i ); ii.hashCode()
  - ❖ Similarly, you need to “unwrap” an Object to use primitive operation.  
int j = ii.intValue() \* 7;



# Autoboxing continued

- ❑ Java 1.5 makes this automatic:

- ❖ Before: Example1

- ```
ArrayList list = new ArrayList();  
list.add(0, new Integer(42));  
int total = (list.get(0)).intValue();
```

- ❖ After: Example2

- ```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, 42);  
int total = list.get(0);
```

# Enumerations

- ❑ An enumeration, or “enum,” is simply a set of constants to represent various values.
- ❑ Here's the old way of doing it:
  - ❖ `public final int SPRING = 0;`  
`public final int SUMMER = 1;`  
`public final int FALL = 2;`  
`public final int WINTER = 3;`
- ❑ This is a nuisance, and is error prone as well.
- ❑ Here's the new way of doing it:
  - ❖ `enum Season { winter, spring, summer, fall }`

# Enumerated Types

- ❑ Similar to enum types in C, Pascal etc. and HashTable, keyset.
- ❑ Enums are classes, extends `java.lang.Enum`.
- ❑ Enums are final, can't be subclassed.
- ❑ Only one Constructor and is protected.
- ❑ Implement `java.lang.Comparable`: `compareTo()`
- ❑ Implement `Serializable`.

# Advantages of the new enum

- ❑ **Type safety:** Enums provide compile-time type safety and prevent from comparing constants in different enums.
- ❑ **Limits inputs:** The compiler won't allow parsing a constant of a wrong enum to a method that expects an enum type. When using enum in switch statement, the enum constants are the only allowed labels for cases. IDEs like eclipse even provide autocomplete feature for filling them. When using enum in switch statement, the enum constants are the only allowed labels for cases. IDEs like eclipse even provide auto complete feature for filling them.
- ❑ Enums **groups things** in a set.
- ❑ Enums are **iterable**.
- ❑ Because they're objects, you can put them in collections.
- ❑ Because they're essentially classes, you can add arbitrary fields and methods.

# Example of enum

```
1. enum Season
2. {
3.     WINTER, SPRING, SUMMER, FALL
4. }
5. class EnumExample
6. {
7.     public static void main(String as[])
8.     {
9.         Season s=Season.SPRING;
10.        System.out.println(s);
12.    }
13. }
```

Output:  
SPRING

# Var-args

- ❑ To declare a method using var-args parameter, you follow the type with an ellipsis(...), a space, and then the name of the parameter that will hold the value received.
- ❑ Allows an unlimited number of parameters for a method.
- ❑ The parameters must all be the same type.
- ❑ The number arguments may be 0, one or more.
- ❑ You can mix var-args and regular arguments.
- ❑ Var-args must be the last parameter in the method's signature, and you can have only one var-args in a method.
- ❑ Varargs are never null.
- ❑ Compiler interprets var-args like an array. Therefore accessing the var-args is like accessing array elements. Either for loop or enhanced for-loop could be used to iterate through var-args.

## Example: Var-args

```
1. public void myMethod(String... args) {  
2.   for (int i = 0; i < args.length; i++) {  
3.     System.out.println("The value at position " + i + " is " + args[i]); }  
4. }
```

Same as:

```
5. public void myMethod(String[] args) {  
6.   for (int i = 0; i < args.length; i++) {  
7.     System.out.println("The value at position " + i + " is " + args[i]); }  
8. }
```

## Example: Var-args continued

```
myMethod("a");           \\Line1
```

```
myMethod("a", "b", "c", "d", "e");  \\Line2
```

```
myMethod();              \\Line3
```

Easier than writing:

```
myMethod(new String[]{"a", "b", "c", "d", "e"});
```



# Scanner class

- ❑ Prior to Java 1.5 getting input from the console involved multiple steps.
- ❑ Java 1.5 introduced the Scanner class which simplifies console input. It can also be used to read from files and Strings (among other sources).
- ❑ It also can be used for powerful pattern matching – we will not cover pattern matching in this class.
- ❑ Scanner is in the Java.util package. So you must:

**import java.util.Scanner;**

- ❑ Constructors:
  - ❖ **Scanner**(File source)  
Constructs a new Scanner that produces values scanned from the specified file.
  - ❖ **Scanner**(InputStream source)  
Constructs a new Scanner that produces values scanned from the specified input stream.
  - ❖ **Scanner**(Readable source)  
Constructs a new Scanner that produces values scanned from the specified source.
  - ❖ **Scanner**(String source)  
Constructs a new Scanner that produces values scanned from the specified string.

# Methods of Scanner class

- ❑ Finally, Java has a fairly simple way to read input.
  - ❖ `Scanner sc = new Scanner(System.in);`
  - ❖ `boolean b = sc.nextBoolean();`
  - ❖ `byte by = sc.nextByte();`
  - ❖ `short sh = sc.nextShort();`
  - ❖ `int i = sc.nextInt();`
  - ❖ `long l = sc.nextLong();`
  - ❖ `float f = sc.nextFloat();`
  - ❖ `double d = sc.nextDouble();`
  - ❖ `String s = sc.nextLine();`
- ❑ By default, whitespace acts as a delimiter, but you can define other delimiters with regular expressions.

## Scanner class continued

- ❑ Scanner will read a line of input from its source (our examples will be from System.in but we have already seen other sources are possible).
- ❑ Example:

```
Scanner sc = new Scanner (System.in);  
  
int i = sc.nextInt();  
  
System.out.println("You entered" + i);
```
- ❑ This example reads a single int from System.in and outputs it to System.out. It does not check that the user actually entered an int.
- ❑ InputMismatchException: This exception can be thrown if you try to get the next token using a next method that does not match the type of the token.

## hasNext methods of Scanner class

- ❑ **boolean hasNext()**  
Returns true if this scanner has another token in its input.
- ❑ **boolean hasNextBoolean()**  
Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
- ❑ **boolean hasNextByte()**  
Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the nextByte() method.
- ❑ **boolean hasNextDouble()**  
Returns true if the next token in this scanner's input can be interpreted as a double value using the nextDouble() method.

## hasNext methods of Scanner class continued

- ❑ **boolean hasNextFloat()**  
Returns true if the next token in this scanner's input can be interpreted as a float value using the `nextFloat()` method.
- ❑ **boolean hasNextInt()**  
Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the `nextInt()` method.
- ❑ **boolean hasNextLine()**  
Returns true if there is another line in the input of this scanner.
- ❑ **boolean hasNextLong()**  
Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the `nextLong()` method.
- ❑ **boolean hasNextShort()**  
Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the `nextShort()` method.

## Another example

```
Scanner sc = new Scanner (System.in);
System.out.print ("Enter first int: ");
while (sc.hasNextInt())
{
    int i = sc.nextInt();
    System.out.println("You entered " + i);
    System.out.print ("Enter another int: ");
}
```

## Module 2: Objectives

After completion of this module, you should be able to:

- ❑ Explain the limitations of array.
- ❑ Understand Java framework hierarchy.
- ❑ Interpret how to use iterator interface to traverse a collection.
- ❑ Set interface, HashSet, LinkedHashSet and TreeSet.
- ❑ List interface, ArrayList, and LinkedList.
- ❑ Explain vector and stack.
- ❑ Define map, HashMap, LinkedHashMap and TreeMap.
- ❑ Differentiate between collections and array classes.

# Limitations of array

- ❑ Arrays stores **similar data** types. That is, array can hold data of same data type values.
- ❑ Once array is created, its **size is fixed**. That is, at runtime if required, more elements cannot be added. This is another limitation of arrays compared to other data structures.
- ❑ Inadequate support for
  - ❖ inserting,
  - ❖ deleting,
  - ❖ sorting, and
  - ❖ searching operations



# Generics

- ❑ Prior to java1.5, collection methods used **Object** in their collection classes.
- ❑ From java1.5 onwards, Java has added newer syntax to allow programmers to create type-safe collections.
- ❑ The type that will be used to create the collection object is specified at the time of instantiation.
- ❑ The collection methods therefore use generic symbols (like 'E').
- ❑ Note that E can represent only class type not primitive type.

Only flowers can go in the flower vase



# Generics

- ❑ It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.
- ❑ Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- ❑ Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- ❑ Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

# Generics Method

- ❑ You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods –
  - ❑ All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
  - ❑ Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
  - ❑ The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
  - ❑ A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

# Example

```
public class GenericMethodTest
{
    public static < E > void printArray( E[] inputArray )
    {
        for(E element : inputArray)
        {
            System.out.println(element); }
        }
    }
}
```

# Bounded Type Parameters

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what bounded type parameters are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound.

# Generic Classes

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

# Collections

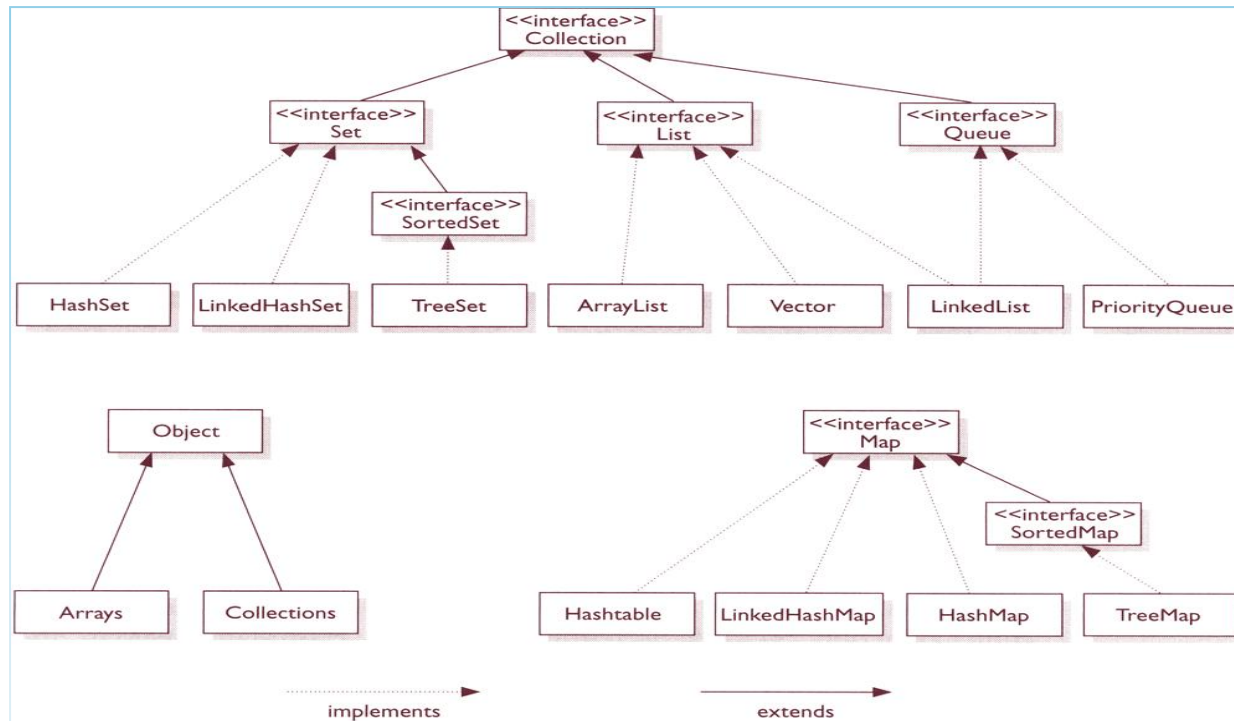
- ❑ A collection in java is an object that can hold multiple objects (like an array). These objects are called the elements of the collection.
- ❑ Collections can grow to any size unlike arrays.
- ❑ A collection framework is a common architecture for representing and manipulating all the collections. This architecture has a set of interfaces on the top and implementing classes down the hierarchy. Each interface has specific purpose.
- ❑ Java Collection Framework supports two types of collections named collections maps.
- ❑ Collection framework uses generics.

# Collections Framework

- ❑ The collections framework was designed to meet several goals.
  - ❖ The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
  - ❖ The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - ❖ Extending and/or adapting a collection had to be easy.
- ❑ A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:
  - ❖ **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
  - ❖ **Implementations i.e. Classes:** Classes which provide implementations of the Interfaces.
  - ❖ **Algorithms:** Algorithms that provide behaviors commonly required when using collections i.e. search, sort, iterate, etc.



# Collections Framework Hierarchy



# Collection Interface

- ❑ It is the root interface in the collection hierarchy.
- ❑ The items in the collection is referred to as elements.
- ❑ **Collection** interface extends another interface called **Iterable**.
- ❑ Any class that implements **Iterable** can use the enhanced for-loop to iterate through elements.
- ❑ **Iterable** has single methods that returns **Iterator**.

**Iterator<T> iterator()**

- ❑ **Iterator** is an interface with 3 methods:
  - ❖ **boolean hasNext()**
  - ❖ **E next()**
  - ❖ **void remove()**

## Collection Interface continued

- ❑ **List** is a collection of objects.
- ❑ **Set** is a collection of objects that does not allow duplicate objects.
- ❑ **Queue** is a collection of objects that arranges objects in FIFO order by comparing the objects and has queue like methods.
- ❑ **Map** contains pairs of objects (each pair comprising of one object representing a key and other representing a value ).
- ❑ Note that hierarchy for **Collection** and hierarchy for **Map** both are part of collection framework.
- ❑ There are 14 collection interfaces.

# Methods in collection interface

- ❑
  - +add(element: Object): boolean
  - +addAll(collection: Collection): boolean
  - +clear(): void
  - +contains(element: Object): boolean
  - +containsAll(collection: Collection): boolean
  - +equals(object: Object): boolean
  - +hashCode(): int
  - +isEmpty(): boolean
  - +iterator(): Iterator
  - +remove(element: Object): boolean
  - +removeAll(collection: Collection): boolean
  - +retainAll(collection: Collection): boolean
  - +size(): int
  - +toArray(): Object[]
  - +toArray(array: Object[]): Object[]

## Most commonly used methods in collection interface

- ❑ **boolean add(E o):** ensures that element is in the collection. For non-duplicate collection it returns false if the element is already in the collection. This is an optional operation.
- ❑ **boolean addAll(Collection<? extends E> c):** adds all of the elements in the specified collection to this collection.
- ❑ **void clear():** deletes all elements in the collection.
- ❑ **boolean remove(Object o):** removes a single instance of the specified object 'o' from this collection, if it is present.
- ❑ **boolean removeAll( Collection<?> c):** removes all the elements specified in the collection 'c' .
- ❑ **boolean retainAll( Collection<?> c):** removes all the elements NOT specified in the collection 'c'.
- ❑ **int size():** returns the size of the collection.
- ❑ **boolean contains(Object o):** returns true if this collection contains the specified element.
- ❑ **boolean containsAll(Collection c):** returns true if this collection contains all of the elements in the specified collection.
- ❑ **boolean equals(Object o):** Compares the specified object with this collection for equality.

# List Interface

- ❑ The List interface extends the Collection interface.
- ❑ A list allows duplicate elements in a collection .
- ❑ Where duplicate elements are to be stored in a collection, list can be used.
- ❑ The subclasses of **List** are ordered collection of objects.
- ❑ The elements are ordered based on user's input.
- ❑ Methods of **List** interface provide indexed access to the objects.
- ❑ The index begins from 0.
- ❑ From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.
- ❑ **List** is also called sequence.

# ArrayList

- ❑ The **ArrayList** class extends **AbstractList** and implements the **List** interface.
- ❑ It can grow dynamically.
- ❑ It maintains insertion order of elements.
- ❑ It allows duplicate object.
- ❑ It provides more powerful insertion and search mechanisms than array.
- ❑ it allows the programmer to traverse the list in either direction and modify the list during iteration.
- ❑ Constructors:
  - ❖ **ArrayList()**  
Creates an array with default capacity of 10.
  - ❖ **ArrayList(int initialCapacity)**  
Creates an array with initial capacity as specified by “initialCapacity”.

# Example of ArrayList

```
1. import java.util.*;
2. class TestArrayList {
3.     public static void main(String as[]) {
4.         ArrayList arrayList = new ArrayList(); → We can add any type of object value in this ArrayList
5.         arrayList.add(new Integer(1));
6.         arrayList.add("Hello");
7.         arrayList.add(10.23);
8.         arrayList.add("Hello");
9.         arrayList.add(true);
10.        System.out.println(arrayList);
11.    }}
```

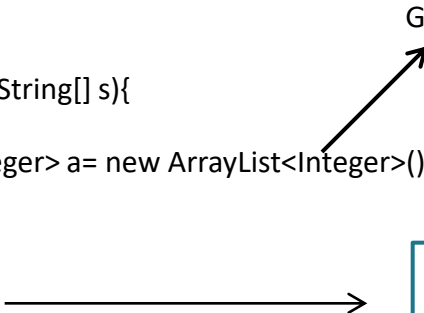
*Output:*

[1,Hello,10.23,Hello,true]



## Example of ArrayList using generics

```
1. import java.util.*;
2. class TestArrayList{
3.     public static void main(String[] s){
4.         ArrayList<Integer> a= new ArrayList<Integer>();
5.         a.add(1);
6.         a.add(2);
7.         a.add(3);
8.         for(Integer o:a)
9.             System.out.println(o);
10. }}
```



Generics Type

We can add only integer type of value into this ArrayList because this is type safe with integer.

# Iterator and enhanced for loop

- Iteration can be done using enhanced for-loop.

```
for(Integer o:a)           //Line1
```

```
System.out.println(o);     //Line2
```

- This is converted by compiler to the code using Iterator. (which is pre 1.5 way of iterating collections)

```
Iterator<Integer> i = a.iterator();           //Line3
```

```
while (i.hasNext())                           //Line4
```

```
System.out.println(i.next());                 //Line5
```

## Traditional way(without generics)

- ❑ Note that the compiler just gives a warning if generics are not used. In eclipse you can see a yellow line.
- ❑ The code below does not use generics. Hence the **ArrayList** can hold any type of object.  
**ArrayList li=new ArrayList(); //Line1**
- ❑ Therefore while using enhanced for-loop/ or any other loop, we can retrieve objects from the collection as **Object** type.
- ❑ We then need to cast the **Object** based on the their types.
- ❑ The compiler does not check if the cast is the same as the actual object type in collection, so the cast can fail at run time.

# Example

```
import java.util.ArrayList;
public class ArrayListEx {
    public static void main(String[] s){
        ArrayList a= new ArrayList();
        a.add(1);
        a.add(1.78);
        double sum=0;
        for(Object o:a) {
            Number d=null;
            // cast the object based on type and use it
            if(o instanceof Number){
                d=(Number)o;
                sum =sum+d.doubleValue();
            }
            System.out.print(sum);
        }
    }
}
```

When we compile and run this code.

1. Warning generated by compiler.
2. Need to cast objects to appropriate types .

Because this ArrayList is not type safe. We can add any type of object into this ArrayList.

## Advantage: Generic way

```
1. import java.util.ArrayList;
2. public class ArrayListExGen {
3.     public static void main(String[] s) {
4.         ArrayList<Number> a= new ArrayList<Number>();
5.         a.add(1);
6.         a.add(1.78);
7.         double sum=0;
8.         for(Number o:a){
9.             sum=sum+o.doubleValue();
10.        }
11.        System.out.print(sum);
12. }}
```

Reduced and safe code. Improved readability and robustness.  
No possibility of unsafe cast.

# Tell me if they are same?

- ❑ Is an instance of **ArrayList<Object>** same as instance of **ArrayList** ?
  - ❖ In terms of what both the instances hold-they are same.
  - ❖ The JRE does not know anything about generics.
  - ❖ Generics is processed at the compilation level. The compiler makes sure that objects that are added to the collection are of valid type.
  - ❖ After the check is done compiler does something called type erasure . (next slide)
  - ❖ For runtime system, **ArrayList<Object>** and of **ArrayList** both are same.
  - ❖ At compiler level, **ArrayList<Object>** requires more compiler intervention. For **ArrayList** only a warning is generated by the compiler.

# Type erasure

- ❑ Generic type information is present only at compile time.
- ❑ After compiler ensures all the checks are met, it *erases* all the generic information.
- ❑ As a result the code looks like the traditional code (like code without generics that used raw type). Which means that the **ArrayList<Integer>** becomes **ArrayList** at runtime.
- ❑ This is done to ensure binary compatibility with older Java libraries and applications that were created before generics.

# Mixing generic and non-generic collections

- ❑ It is not recommended to mix generics and legacy code.
- ❑ But there are time when we need to do it in cases where we need to pass collection arguments to legacy code and vice versa. In such case we need to convert generics to raw type and vice versa in such cases.
- ❑ Following are the legal ways of mixing generics to raw types:

1. **`ArrayList<Integer> a= new ArrayList();`**
2. **`ArrayList a= new ArrayList<Integer>();`**
3. **`List<Integer> a= new ArrayList();`**
4. **`List a= new ArrayList<Integer>();`**

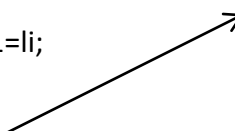


## Problem with mixing generic and non-generic collections

Example 1:

```
1. import java.util.*;
2. class Test{
3.     public static void main(String[] s){
4.         ArrayList li=new ArrayList();
5.         li.add(1);
6.         li.add("Hello");
7.         ArrayList<Integer> li1=li;
8.         for(Integer i:li1)
9.             System.out.println(i);
10. }}
```

java.lang.ClassCastException: java.lang.String cannot be  
cast to java.lang.Integer at runtime.



Example 2:

```
List l= new ArrayList<Integer>();
l.add("AbC");
```

# Polymorphism and Generics

- ❑ **List<Integer> a= new ArrayList<Integer>();**
- ❑ is valid but
- ❑ **List<Number> a= new ArrayList<Integer>();**
- ❑ is compilation error
- ❑ If compiler allowed it, then it would be possible to insert a Number that is not a Integer into it, which violates type safety. And remember we are using generics precisely because we want type safe collections!

## Wild card characters with generics

- ❑ **Y<? extends X>**
- ❑ **Y<? super X>**
- ❑ **Y<?>**
- ❑ Y represents any collection class and X represents any class or interface.
- ❑ Wild card characters can be used only on the left-hand-side of the assignment statement.

## Y<? extends X>

- ❑ This syntax allows all objects of type X (that instances of X and its subclasses of X) to be in the collection.
- ❑ Also reference of this type cannot be used to add objects in the collection.
- ❑ Example:
  - ❖ **`ArrayList<? extends Number> l = new ArrayList<Integer>();`**
  - ❖ **`ArrayList<? extends Number>`** references can be used to hold **`Number`** or any subclass of **`Number`**.
  - ❖ cannot be used for adding elements.

## Example: Y<? extends X>

```
1. import java.util.*;
2. class Test{
3.     public static void main(String[] s){
4.         ArrayList<Integer> li=new ArrayList<Integer>();
5.         li.add(1);
6.         li.add(2);
7.         ArrayList<? extends Number> m= li;
8.         m.add(1);                // Generates compilation error
9.         m.remove(0);             // It will Compile fine
10.        System.out.println(m.get(1));    //It will Compile fine
11.    }}
```

## Y<? super X>

- ❑ This syntax allows all objects of type X and its super class types to be in the collection.
- ❑ **ArrayList<? super Integer>** reference can hold any elements of type **Integer** or super class of **Integer**.
- ❑ Allows all the operations including **add**.
  1. **ArrayList<? super Integer> li=new ArrayList<Integer>();**
  2. **li.add(1);**
  3. **li.add(2);**
  4. **System.out.println(li.get(1));**

## Y<?>

- ❑ This is similar to **<? extends Object>**.
- ❑ A reference of **ArrayList<?>** can hold any type of **Object** but cannot be used for adding elements.

```
1. ArrayList<Integer> li=new ArrayList<Integer>();  
2. li.add(1);  
3. li.add(2);  
4. ArrayList<?> m=li;  
5. m.add(1); // Generates compilation error  
6. System.out.println(m.get(1)); // It will compile fine
```

## Conversion with generics

- ❑ 1. `ArrayList<Object> a= new ArrayList<Student>();` `// Compilation error`
- 2. But `ArrayList<Object> a= new ArrayList<Object>();`
- 3. `a.add(new Student("Raj"));` `// It will compile`
- ❑ 4. `ArrayList<? extends Object> a= new ArrayList<Student>();` `// It will compile`
- ❑ 5. `ArrayList<? super Student> a= new ArrayList<Student>();` `// It will compile`
- 6. `a.add(new Student("Raj"));` `// It will compile`
- ❑ 7. `ArrayList<?> a= new ArrayList<Student>();` `//It will compile`



## Back to List classes - LinkedList

- ❑ The **LinkedList** class extends **AbstractSequentialList** and implements the **List** and **Queue** interface.
- ❑ All the **List** classes we have seen so far used arrays internally. **LinkedList** class uses doubly-linked list.
- ❑ LinkedList provides support for random access through an index with inserting and deletion elements from any place . It allows duplicate object.
- ❑ Constructors:
  - ❖ **LinkedList()**
  - ❖ **LinkedList(Collection c)**

# Example of LinkedList

```
1. import java.util.*;
2. class TestLinkedList {
3.     public static void main(String as[]) {
4.         LinkedList<String> linkedList = new LinkedList<String>();
5.         linkedList.add("A");
6.         linkedList.add("B");
7.         linkedList.add("D");
8.         linkedList.removeLast();    //[A, B]
9.         linkedList.addFirst("E");    //[E, A, B]
10.        System.out.println(linkedList); }}
```

*Output:*

[E, A, B]

# ArrayList vs. LinkedList

- ❑ ArrayList provides support random access through an index without inserting or removing elements from any place other than an end.
- ❑ LinkedList provides support for random access through an index with inserting and deletion elements from any place .
- ❑ If your application does not require insertion or deletion of elements, the Array is the most efficient data structure.

## What to choose and when: **ArrayList** or **LinkedList**

- ❑ **ArrayList** uses arrays. When your application needs to randomly access the elements in the list. Calling `get()` methods using index will be faster in case of an array than linked list.
- ❑ On the other hand if application has to add random amount of data or add data at random positions, then **LinkedList** class is preferred.

# Vector class

- ❑ The **Vector** class is exactly same as **ArrayList** class except that the Vector class methods are thread-safe.
- ❑ Thread-safe means the most of the methods of **Vector** class have **synchronized** keyword. Hence no 2 threads can access the same instance of **Vector** class simultaneously if both are accessing **synchronized** methods.
- ❑ Having thread-safe code is good but sometimes in applications we might not be need thread-safety. In such cases **synchronized** code might be an overburden making the execution slow.
- ❑ Constructor
  - Vector()**
  - Vector(int initialCapacity)**
  - Vector(int initialCapacity, int capacityIncrement)**

# Stack class

- ❑ The Stack class represents a last-in-first-out (LIFO) stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.
- ❑ Inherits from the **Vector** class.
- ❑ Constructor :
  - ❖ **Stack()**
- ❑ Methods
  - ❖ +empty() : boolean
  - ❖ +peek() : Object
  - ❖ +pop() : Object
  - ❖ +push(element: Object) : void
  - ❖ +search(element: Object) : int

## Example: Stack

```
1. import java.util.*;
2. public class ArrayListExG {
3.     public static void main(String[] s){
4.         Stack<Character> l=new Stack<Character>();
5.         l.push('a');
6.         l.push('b');
7.         l.push('c');
8.         System.out.println(l.peek());
9.         System.out.println(l.search('a'));
10. }}
```

Output:

c  
3

**Note**→ **push** method is replaced by **add**, we still will get the same result.

# Set Interface

- ❑ The Set interface extends the Collection interface.
- ❑ The Set interface is used to represent a collection which does not contain duplicate elements.
- ❑ The classes that implement Set must ensure that no duplicate elements can be added to the set.
- ❑ But how will we determine duplicates objects?
  - ❖ Two objects **o1** and **o2** are duplicates if **o1.equals(o2)** returns **true**. That is, a **Set** cannot contain both **o1** and **o2** such that **o1.equals(o2)** is **true**.
- ❑ It can contain at most one **null** element.
- ❑ No new methods or constants apart from what it gets from **Collection** interface.
- ❑ Classes implementing **Set** must:
  - ❖ return **false** if an attempt is made to add duplicate element.



# HashSet

- ❑ This is an unsorted, unordered Set. This may be chosen when order of the elements are not important.
- ❑ Also there is no guarantee that the order will remain constant over time when new entries are added.
- ❑ **HashSet** stores its elements in a hash table.
- ❑ Therefore this class offers constant time performance for the basic operations like **add**, **remove**, **contains** etc.
- ❑ This class relies heavily on **hashCode()** method of the object that is added in **HashSet**.
- ❑ Constructors
  - ❖ HashSet()
  - ❖ HashSet(int initialCapacity)
  - ❖ HashSet(int initialCapacity, float loadFactor)
  - ❖ HashSet(Collection<? extends E> c)

## Example: HashSet

```
1. import java.util.*;
2. class TestLinkedList {
3.     public static void main(String as[]) {
4.         Set<String> s = new HashSet<String>();
5.         s.add("A");
6.         s.add("B");
7.         s.add("A"); s.add("D");
8.         System.out.println(s);
9.         System.out.println("Size of Set="+s.size());}}
```

*Output:*[A,D,B]

Size of Set=3

# LinkedHashSet

- ❑ Subclass of **HashSet**, maintains the insertion-order and does not allow duplicates.
- ❑ If a duplicate element is entered, insertion order of the first one is maintained since 2<sup>nd</sup> one is not inserted at all.
- ❑ It implements a hashtable using doubly-linked list.
- ❑ Like **HashSet**, this class also has constant-time performance for the basic operations (add, contains and remove) if the hash function is implemented properly. But compared to **HashSet**, this class is slow except in case of iterating over the collection in which case **LinkedHashSet** is faster.
- ❑ Same constructor and methods like **HashSet**.
- ❑ Like **HashSet** this is also not a thread-safe class.

## Example: LinkedHashSet

```
import java.util.*;

class TestLinkedList {

    public static void main(String as[]) {

        Set<String> s = new LinkedHashSet <String>();
        s.add("A");

        s.add("B");

        s.add("A"); s.add("D");

        System.out.println(s);

        System.out.println("Size of Set="+s.size());}}
```

*Output:*[A,B,D]

Size of Set=3

# TreeSet

- ❑ This is a sorted set. The elements in this will be in ascending order in the natural order. You can also define a custom order by means of a `Comparator` passed as a parameter to the constructor.
- ❑ `SortedSet` is a subinterface of `Set`, which guarantees that the elements in the set are sorted.
- ❑ **TreeSet** implements **NavigableSet** interface which extends **SortedSet** interface.
- ❑ The elements can be sorted in two ways.
  - ❖ One way is to use the `Comparable` interface. Objects can be compared using by the `compareTo()` method. This approach is referred to as a natural order.
  - ❖ The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the `Comparable` interface, or you don't want to use the `compareTo()` method in the class that implements the `Comparable` interface. This approach is referred to as order by comparator.
- ❑ This is not a thread-safe class.

## Example: TreeSet

```
import java.util.*; //Line1
class TestTreeSet{ //Line2
    public static void main(String as[]){ //Line3
        Set<String> hashSet = new HashSet<String>(); //Line4
        hashSet.add("Yellow"); //Line5
        hashSet.add("White "); //Line6
        hashSet.add("Green"); //Line7
        hashSet.add("Orange"); //Line8
        System.out.println("An unsorted set of strings"); //Line9
        System.out.println(hashSet + "\n"); //Line10
        Set<String> treeSet = new TreeSet<String>(hashSet); //Line11
        System.out.println("Sorted set of strings"); //Line12
        System.out.println(treeSet + "\n"); //Line13
    }
}
```

### Output:

An unsorted set of strings  
[Orange, Green, White, Yellow]  
A sorted set of strings  
[Green, Orange, White, Yellow]

# TreeSet continued

## ❑ Methods (from **SortedSet**)

- ❖ **Object first()**
- ❖ **Object last()**
- ❖ **Comparator<? super E> comparator()**
- ❖ **SortedSet<E> subSet(E fromElement, E toElement)**
- ❖ **SortedSet<E> headSet(E toElement)**

Returns portion of the set whose elements are < toElement.

- ❖ **SortedSet<E> tailSet(E fromElement)**

Returns a portion of the set whose elements are >= to fromElement.

## TreeSet continued

### ❑ Methods (from **NavigableSet**)

- ❖ **E ceiling(E e)**

- ❖ **E floor(E e)**

- ❖ **E higher(E e)**

- ❖ **E lower(E e)**

Returns the element in the set  $\geq$ ,  $\leq$ ,  $>$  or  $<$  to the given element, respectively or null if there is no such element.

- ❖ **E pollFirst()**

- ❖ **E pollLast()**

Retrieves and removes the first (lowest)/ the last (highest) element, respectively or returns null if this set is empty.



# Interface Queue

- ❑ Extends Collection.
- ❑ Designed for holding elements prior to processing.
- ❑ Typically ordered in a FIFO manner.
- ❑ Methods in this interface:
  - ❖ To add an element in a queue
    - **boolean offer(E o)**
  - ❖ To remove an element from the queue:
    - **E poll()** : returns **null** if called on empty Queue
    - **E remove()**: throws **NoSuchElementException** if called on empty **Queue**
  - ❖ To retrieves but nor remove an element from the queue:
    - **E peek()**: returns **null** if called on empty Queue
    - **E element()** throws **NoSuchElementException** if called on empty **Queue**

# PriorityQueue

- ❑ It is a sorted collection that implements **Queue**.
- ❑ Sorting is based on:
  - ❖ the **compareTo()** method of **Comparable** interface(also called *natural order*). When natural order is used, if the elements added to the **PriorityQueue** is not of **Comparable** type, a **ClassCastException** is thrown at runtime.
  - ❖ the **Comparator** object passed via constructor. For elements that do not implement **Comparable** and for elements that implement **Comparable** but the ordering in **compareTo()** is not what is desired, this can be used.
- ❑ A **PriorityQueue** does not permit **null** elements.
- ❑ Note that this class is not thread-safe.

# PriorityQueue continued

## ❑ Constructors

- ❖ **PriorityQueue()**

- ❖ **PriorityQueue(int initialCapacity)**

Creates a **PriorityQueue** and orders its elements based on natural ordering.

- ❖ **PriorityQueue(int initialCapacity, Comparator<? super E> comparator)**

Creates a **PriorityQueue** and orders its elements based on **Comparator** instance passed.

## ❑ Methods (new added here)

- ❖ **Comparator<? super E> comparator()**

## Example: PriorityQueue

```
1. import java.util.*;
2. public class PQExample {
3.     public static void main(String as[]){
4.         PriorityQueue<String> p= new PriorityQueue<String>();
5.         p.offer("B");
6.         p.offer("D");
7.         p.offer("A");
8.         p.offer("C");
9.         while(!p.isEmpty()) {
10.             String s=p.poll();
11.             System.out.println(s ); }
12. }
```

Output:

A  
B  
C  
D

# Map Interface

- ❑ A Map is a storage that maps keys to values. There cannot be duplicate keys in a Map and each key maps to at most one value.
- ❑ The Map interface is not an extension of Collection interface. Instead the interface starts of it's own interface hierarchy.
- ❑ for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition.
- ❑ The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.
- ❑ Note that **Map** is not **Iterable**, therefore enhanced for loop cannot be used for **Map**.

# Map Interface continued

## ❑ Methods:

- ❖ `boolean containsKey(Object key)`

- ❖ `boolean containsValue(Object value)`

Returns true if map contains specified key (1<sup>st</sup> method) or specified value (2<sup>nd</sup> method)

- ❖ `void clear()`

Removes all mappings from this map (optional operation).

- ❖ `Set<Map.Entry<K,V>> entrySet()`

Returns a set view of the mappings contained in this map.

- ❖ `boolean equals(Object o)`

Compares the specified object with this map for equality.

- ❖ `V get(Object key)`

Returns the value to which this map maps the specified key.

## Methods continued

- ❑ `boolean isEmpty()`  
Returns true if this map contains no key-value mappings.
- ❑ `Set<K> keySet()`  
Returns a set view of the keys contained in this map.
- ❑ `V put(K key, V value)`  
Associates the specified value with the specified key in this map (optional operation).
- ❑ `void putAll(Map<? extends K, ? extends V> t)`  
Copies all of the mappings from the specified map to this map (optional operation).
- ❑ `V remove(Object key)`  
Removes the mapping for this key from this map if it is present (optional operation).
- ❑ `int size()`  
Returns the number of key-value mappings in this map.

# HashMap , TreeMap and Hashtable

- ❑ The HashMap and TreeMap classes are two concrete implementations of the Map interface. Both will require key & value. Keys must be unique.
- ❑ The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.
- ❑ The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.
- ❑ HashMap allows null as both keys and values.
- ❑ TreeMap is slower than HashMap.
- ❑ TreeMap allows us to specify an optional Comparator object during its creation. This comparator decides the order by which the keys need to be sorted.
- ❑ Hashtable is same like HashMap . The only difference between HashMap and Hashtable is that Hashtable is thread-safe.



## Example: Map interface

```
1. import java.util.*;
2. class MapEx{
3.     public static void main(String args[]) {
4.         Map<Integer,String> map = new HashMap<Integer,String>();
5.         map.put (1, "one");
6.         map.put (2, "two");
7.         System.out.println(map.size() );           //print 2
8.         System.out.println(map.get("1") );         //print "one"
9.         Set keys = map.keySet();
10.        for (Object object : keys) {    System.out.println(object);  }           // print the keys
        }
    }
```

## Example: TreeMap

```
class TreeMapTest
{
    public static void main(String[] s){
        TreeMap<String,Double> hm = new
        TreeMap<String,Double>();
        // Put elements to the map
        hm.put("A", new Double(10000.00));
        hm.put("C", new Double(5000.22));
        hm.put("D", new Double(7000.00));
        hm.put("B", new Double(4000.00));
        Set<Map.Entry<String,Double>> set =
        hm.entrySet();
        // Get an iterator
        Iterator<Map.Entry<String,Double>> i =
        set.iterator();
```

```
// Display elements
while(i.hasNext())
{
    Map.Entry<String,Double> me = i.next();
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}
}}
```

### Output:

```
A: 10000.0
B: 4000.0
C: 5000.22
D: 7000.0
```

# Ordering the objects

- ❑ Objects in applications do appear in collections.
- ❑ Object collections in application may need sorting /ordering.
- ❑ The equal method of the Object class does not help in comparing the objects.
- ❑ To provide for ordering/sorting of the objects Java API provides two interfaces Comparable and Comparator.
  - ❖ Comparable interface is in java.lang.
  - ❖ Comparator interface is in java.util.

# Implementing comparable interface

```
public class Employee implements Comparable {  
    String name;  
    int salary;  
    public Employee(String name, int salary) {...}  
  
    public int compareTo(Object o) throws ClassCastException {...}  
  
    public static void main(String args[]) {...}  
}
```

# Constructor for Employee

- ❑ Constructor for Employee—nothing new here.
  - ❖ 

```
public Employee(String name, int salary) {  
    this.name = name;  
    this.salary = salary;  
}
```
- ❑ Sorting Employee according to their salary.
- ❑ Comparisons happen between two objects.
- ❑ Whatever kind of collection they may or may not be in.

## The main method

```
1. public static void main(String args[]) {  
2.     TreeSet<Employee> set = new TreeSet<Employee>();  
  
3.     set.add(new Employee("Ann", 87000));  
4.     set.add(new Employee("Bob", 83000));  
5.     set.add(new Employee("Tom", 97000));  
6.     set.add(new Employee("Dan", 25000));  
7.     set.add(new Employee("Eve", 75000));  
  
8.     Iterator<Employee> iter = set.iterator();  
9.     while (iter.hasNext()) {  
10.         Employee s = iter.next();  
11.         System.out.println(s.name + " " + s.salary);  
    }  
}
```

# Using the TreeSet

- ❑ In the main method we have the line.
  - ❖ `TreeSet set = new TreeSet();`
- ❑ Later we use an iterator to print out the values in order, and get the following result:
  - ❖ Dan 25000
  - ❖ Eve 75000
  - ❖ Bob 83000
  - ❖ Ann 87000
  - ❖ Cat 97000
- ❑ How did the iterator know that it should sort Employee by salary, rather than by name?

## Implementing comparable <T>

- ❑ public class Employee implements Comparable.
- ❑ This means it must implement the method.  
`public int compareTo(Object o)`
- ❑ Notice that the parameter is an Object.
- ❑ In order to implement this interface, our parameter must also be an Object, even if that's not what we want.
- ❑ 

```
public int compareTo(Object o) throws ClassCastException {  
    if (o instanceof Employee)  
        return this.salary - ((Employee)o).salary;  
    else  
        throw new ClassCastException("Not a Employee!");  
}
```
- ❑ A ClassCastException should be thrown if we are given a non-Employee parameter.



# An improved method

- ❑ Since casting an arbitrary Object to a Employee may throw a classCastException for us, we don't need to throw it explicitly:
- ❑ **Example1**
  - ❖ 

```
public int compareTo(Object o) throws ClassCastException {  
    return this.salary - ((Employee)o).salary;  
}
```
- ❑ Moreover, since classCastException is a subclass of RuntimeException, we don't even need to declare that we might throw one:
- ❑ **Example2**
  - ❖ 

```
public int compareTo(Object o) {  
    return this.salary - ((Employee)o).salary;  
}
```
- ❑ We discuss more exceptions in next coming session.

# Using Comparator

- ❑ In the previous program just finished, Employee implemented Comparable:
  - ❖ Employees were sorted only by their salary.
  - ❖ If employees are to be sorted another way, such as by name its not possible.
  - ❖ Because there can be only one method of same signature.
- ❑ The problem is solved by placing comparison method in a separate class that implements Comparator instead of Comparable.
  - ❖ Comparator can be used provide sorting by more than one variable.
  - ❖ Comparator requires a definition of compare and equal methods.
  - ❖ `int compare (T object1, T object2)`  
Compares its two arguments for order.
  - ❖ `boolean equals(Object obj)`  
Indicates whether some other object is "equal to" this comparator.

# Outline of employee comparator

```
1. import java.util.*;
2. public class EmployeeComparator implements Comparator<Employee>
3. {
4.     public int compare(Employee e1, Employee e2) {...}
5.     public boolean equals(Object o1) {...}
6. }
```

- ❑ **Note:** When we are using this Comparator, we don't need the compareTo method in the Employee class.
- ❑ Because of generics, our compare method can take Employee arguments instead of just Object arguments.

# The compare method of Comparator interface

```
public int compare(Employee e1, Employee e2) {  
    return e1.salary - e2.salary;  
}
```

- ❑ This differs from compareTo(Object o) in Comparable in these ways:
  - ❖ The name is different.
  - ❖ It takes both objects as parameters, not just one.
  - ❖ We have to either use generics, or check the type of both objects.
- ❑ If our parameters are Objects, they have to be cast to Employees.

# The main method

- ❑ The main method is just like before, except that instead of  
`TreeSet<Employee> set = new TreeSet<Employee>(); //Line1`
- ❑ We have  
`EmployeeComparator comp = new EmployeeComparator(); //Line2`  
`TreeSet<Employee> set = new TreeSet<Employee>(comp); //Line3`

## When to use each

- ❑ The Comparable interface is simpler and less work.
  - ❖ Your class implements Comparable.
  - ❖ You provide a public `int compareTo(Object o)` method.
  - ❖ Use no argument in your `TreeSet` or `TreeMap` constructor.
  - ❖ You will use the same comparison method every time.
- ❑ The Comparator interface is more flexible but slightly more work.
  - ❖ Create as many different classes that implement Comparator as you like.
  - ❖ You can sort the `TreeSet` or `TreeMap` differently with each.
  - ❖ Construct `TreeSet` or `TreeMap` using the comparator you want.
  - ❖ For example, sort `Employees` by salary or by name.

## Sorting differently

- ❑ Suppose you have employees sorted by salary, in a TreeSet you call employeesBySalary.

- ❑ Now you want to sort them again, this time by name.

```
Comparator<Employee> nameComparator = new EmployeeNameComparator(); //Line1
```

```
TreeSet employeesByName = new TreeSet(nameComparator); //Line2
```

```
employeesByName.addAll(employeesBySalary);
```

# Project

---

Create Account

Show Balance

Deposit

Withdraw

Fund Transfer

Print Transaction





## Module 2: Inner class

# Inner Class

- ❑ Like variables and methods, class can also be defined inside a class.
- ❑ An inner class is a class defined inside the scope of another class.
- ❑ Classes that were covered so far were top-level classes.
- ❑ The class inside which the inner class is defined is called outer class.
- ❑ Inner class can access even the private members of the outer class. Similarly outer class can also access the private members of inner class.
- ❑ Similar to inner class, inner interface can also be created.

## Example: Inner Class

```
class Outer
{
    class Inner{}
}
```

- ❑ Inner class instance has access to all member of the outer class( public , private & protected).
- ❑ Inner class can be access through live instance of outer class.
- ❑ The name of the inner class's .class file name: **Outer\$Inner.class** .

# Types of Inner Class

- ❑ Member class
  - ❖ Static Inner Class/ Top-Level nested classes.
  - ❖ Non Static Inner Class.
- ❑ Local Inner Class.
- ❑ Anonymous Class.
- ❑ Non Static Inner Class, Local Inner Class, Anonymous Class are generally called inner class.
- ❑ Static inner class are considered to be top-level class.

# Non static inner class

- ❑ Structure:

```
public class OuterClass{  
    class InnerClass{..}  
}
```

- ❑ Non static inner class object cannot be created without a outer class instance.
- ❑ The **private** fields and methods of the member classes are available to the enclosing class and other member classes.
- ❑ All the **private** fields and methods of the outer classes are also available to inner class.
- ❑ Non-static inner class cannot have **static** members.
- ❑ Other modifier applicable here are : - **abstract, final, public, protected, private.**

## Example: Non static inner class instance

```
1. class MyOuter
2. {
3.     private int x = 7;
4.     class MyInner
5.     {
6.         public void seeOuter()
7.         {
8.             System.out.println("Outer x is " + x);
9.         }
10.    } // close inner class definition
11.    public static void main(String a[])
12.    {
13.        MyOuter mo = new MyOuter();
14.        MyOuter.MyInner inner = mo.new MyInner();
15.        inner.seeOuter();
16.    }
17. }
```

- ❑ Outside outer class non-static inner class creation requires outer class instance also.

- ❑ There are 2 ways to do this.

- ❖ If you don't need outer class instance , then create it like :

**MyOuter.MyInner inner = new MyOuter.new MyInner();**

- ❖ If you need outer class instance or already have one, then create it like :

**MyOuter mo = new MyOuter();**

**MyOuter.MyInner inner = mo.new MyInner();**

## Outer class implicit reference in inner class

- ❑ Non-static inner class instance cannot exist without Outer class instance.
- ❑ This inner class has implicit reference to the outer class object using which it is created.
- ❑ Therefore no explicit reference required in inner class for the outer class.
- ❑ However, if outer class needs an inner class reference it has to create it explicitly.

# Name conflict

- ❑ If the name of the members in Outer class and inner class are same, then how to refer to the name of the outer class member in the inner class?
- ❑ This can be done using Outer class name dot (.) this dot (.) member name.

```
class Outer {  
    int i;  
    class Inner {  
        int i;  
        void f()  
        {  
            i=10;  
            Outer.this.i=9;  
        }  
    }  
}
```



# Static inner class

- ❑ Static nested classes referred to as top-level nested classes, or static inner classes, but they really aren't inner classes at all, by the standard definition of an inner class.
- ❑ It is simply a non-inner (also called “top-level”) class scoped within another. So with static classes it's really more about name-space resolution than about an implicit relationship between the two classes.
- ❑ A static nested class is simply a class that's a static member of the enclosing class, as follows:

```
class BigOuter {  
    static class Nested { }  
}
```

- ❑ The static modifier in this case says that the nested class is a static member of the outer class. That means it can be accessed, as with other static members, without having an instance of the outer class.
- ❑ It can access only all **static** members of the outer class.

# Instantiating a static inner class

- ❑ The syntax for instantiating a static nested class is a little different from a normal inner class, and looks like this:

```
1. class BigOuter {  
2.     static class Nested { }  
3. }  
4. class Broom {  
5.     public static void main (String [] args) {  
6.         BigOuter.Nested n = new BigOuter.Nested(); //Use both class names  
7.     }  
8. }
```

# Method-Local Inner Classes

- ❑ An inner class that is defined inside a method is called local inner class (or method local inner class).
- ❑ A local inner class can be instantiated only by the method which defined it.
- ❑ Therefore no access specifier is applicable for the local inner class declaration. Only **abstract** and **final** modifiers are allowed.
- ❑ Also like other inner classes, local inner class can access all the members of the outer class including private members.
- ❑ Apart from the above, the local inner class can also access local variables which are **final**.
- ❑ 

```
class OuterClass {  
    void someMethod(){  
        class InnerClass{}  
    }  
}
```

# Example:

□ The following legal code shows how to instantiate and use a method-local inner class:

```
1. class MyOuter2 {  
2.     private String x = "Outer2";  
3.     void doStuff() {  
4.         class MyInner {  
5.             public void seeOuter() {  
6.                 System.out.println("Outer x is " + x);  
7.             }}  
8.         MyInner mi = new MyInner(); // This line must come after the class  
9.         mi.seeOuter(); }  
10.    public static void main(String a[]) {  
11.        MyOuter2 m1=new MyOuter2();  
12.        m1.doStuff();  
13.}} // close outer class
```

Example: using the local variables of the method the inner class is in.

```
class MyOuter2 {  
    private String x = "Outer2";  
    void doStuff() {  
        String z = "local variable";  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("Outer x is " + x);  
                System.out.println("Local variable z is " + z); // Won't Compile!  
            } // close inner class method  
        } // close inner class definition  
    } // close outer class method doStuff()  
} //
```

Marking the local variable **z** as final fixes the problem:  
final String z = "local variable"; // Now inner object can use it

# Anonymous Inner Classes

- ❑ Inner class without a class name is an anonymous inner class.
- ❑ Allows creation of one time use object !
- ❑ Anonymous inner class can be created either inside a method or outside a method. It is implicitly **final**.
- ❑ No modifier is allowed anywhere in the class declaration.
- ❑ Also declaration cannot have an **implements** or **extends** clause.
- ❑ No constructors can be defined.
- ❑ An anonymous inner class is either inherited from an interface or from a class and so polymorphism is applicable. It cannot inherit from more than one class directly.

# Syntax

- General way to create an anonymous inner class:

```
class OuterClass{
```

```
...
```

```
    SomeClassOrInterface obj = new SomeClassOrInterface()
```

```
    {
```

```
        // overridden methods
```

```
    };
```

```
}
```



Note the semicolon here!

## Example: Anonymous Inner Classes

```
class Popcorn {  
    public void pop() {  
        System.out.println("popcorn");  
    }  
}  
  
class Food {  
1.   Popcorn p = new Popcorn() {  
2.       public void pop()  
3.       {  
4.           System.out.println("anonymous popcorn");  
5.       }  
6.   };  
}
```

Let's look at what's in the preceding code:

- ❑ We define two classes, Popcorn and Food.
- ❑ Popcorn has one method, pop().
- ❑ Food has one instance variable, declared as type Popcorn.
- ❑ That's it for Food. Food has no methods. And here's the big thing to get:
- ❑ The Popcorn reference variable refers not to an instance of Popcorn, but to an instance of an anonymous (unnamed) **subclass** of Popcorn.
- ❑ Let's look at just the anonymous class code:

```
Popcorn p = new Popcorn() {  
    public void pop() {  
        System.out.println("anonymous popcorn");  
    }  
};
```



## Inner class in interface and vice versa

- ❑ A class can be nested inside an interface. Though this is allowed in java, it is a bad practice to include implementation inside abstraction.
- ❑ A interface can be nested inside a class (or an interface). This is a very rarely used feature.

# Questions

