OBJECT ORIENTED PROGRAMMING WITH JAVA

Dr. N MEHALA
Department of Computer Science and Engineering

INHERITANCE BASICS

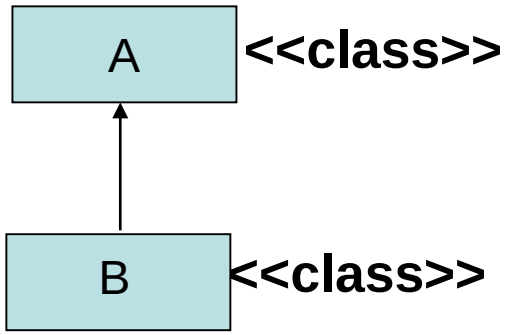
1. **Reusability is achieved by *INHERITANCE***
2. **Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.**
3. *Inheritance* allows a software developer to derive a new class from an existing one
4. **The class whose properties are extended is known as *super or base or parent class*.**
5. **The class which extends the properties of super class is known as *sub or derived or child class***
6. ***A class can either extends another class or can implement an interface***

Forms of Inheritance

class B extends A { ….. }

A super class

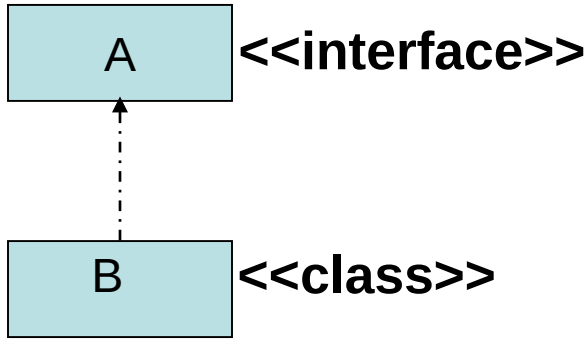
B sub class



class B implements A { ….. }

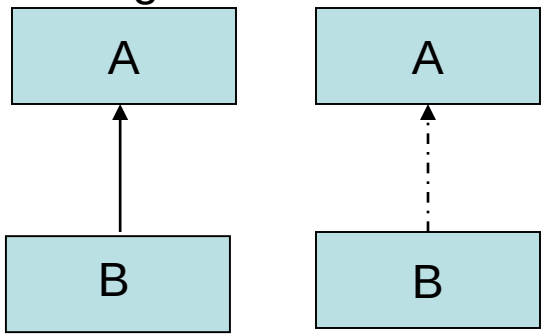
A interface

B sub class

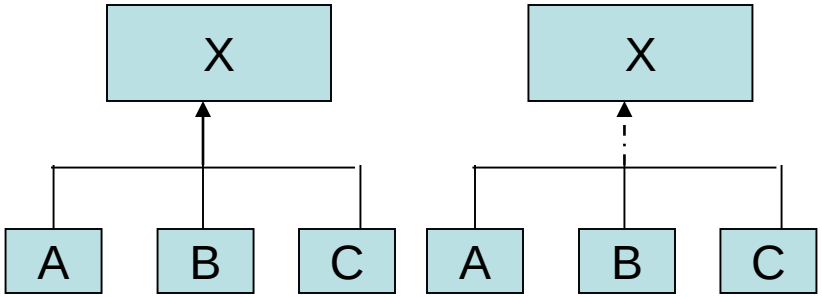


Various Forms of Inheritance

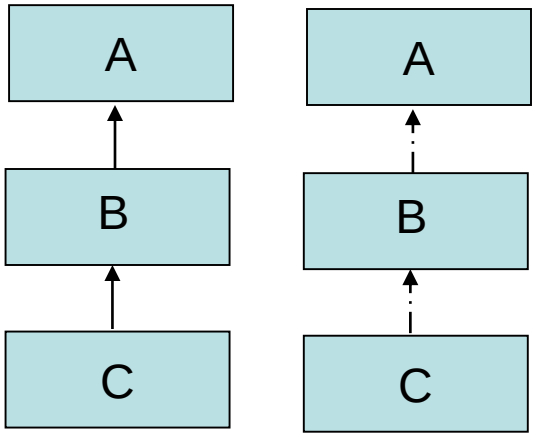
Single Inheritance



Hierarchical Inheritance



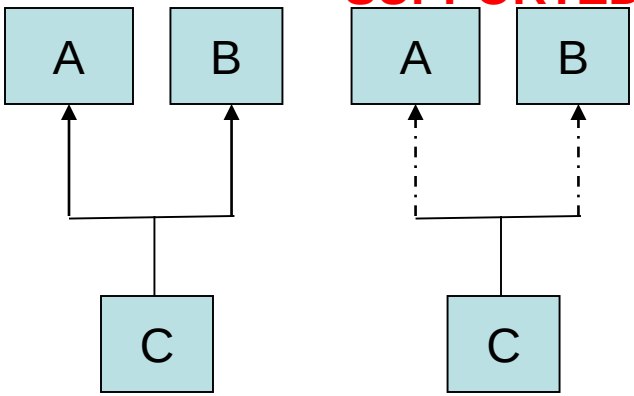
MultiLevel Inheritance



NOT SUPPORTED BY JAVA

Multiple Inheritance

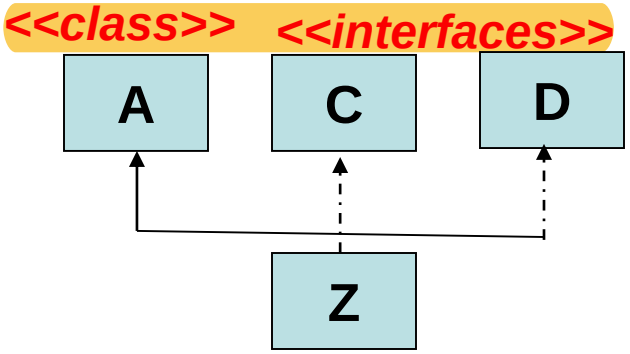
SUPPORTED BY JAVA



An interface can **extend** any number of interfaces but one interface **cannot implement** another interface, because if any interface is implemented then its methods must be defined and interface never has the definition of any method.

Forms of Inheritance

- **Mulitiple Inheritance can be implemented by implementing multiple interfaces not by extending multiple classes**



Example :

```
class Z extends A implements C , D
{ ..........}
```

OK

class A extends B,C
{
}

WRONG

class A extends B extends C
{
}

WRONG

Defining a Subclass

Syntax :

```
class <subclass name> extends <superclass name>
{
 variable declarations;
 method declarations;
}
```

1. Extends keyword signifies that properties of the super class are extended to sub class
2. Sub class will not inherit private members of super class

Access Control

Access Modifiers → / Access Location ↓	public	protected	friendly	private protected	private
Same Class	Yes	Yes	Yes	Yes	Yes
sub classes in same package	Yes	Yes	Yes	Yes	No
Other Classes in Same package	Yes	Yes	Yes	No	No
Subclasses in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

INHERITANCE BASICS

1. Whenever a sub class object is created ,super class constructor is called first.

2. If super class constructor does not have any constructor of its own OR has an unparametrized constructor then it is automatically called by Java Run Time by using call super()

3. If a super class has a parameterized constructor then it is the responsibility of the sub class constructor to call the super class constructor by call

   super(<parameters required by super class>)

4. Call to super class constructor must be the first statement in sub class constructor

INHERITANCE BASICS

When super class has a Unparametrized constructor

```
class A
{
A()
{
System.out.println("This is constructor of class A");
}
} // End of class A
class B extends A
{
B()
{
super();
System.out.println("This is constructor of class B");
}
} // End of class B
```

Optional

Cont…..

INHERITANCE BASICS

```
class inhtest
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

OUTPUT

This is constructor of class A
This is constructor of class B

INHERITANCE BASICS

```
class A
{
A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{System.out.println("This is class B");}
}
class inherit1
{
public static void main(String args[])
{
B b1 = new B();
}   }
```

File Name is xyz.java

```
/*
E:\Java>javac xyz.java

E:\Java>java xyz
Exception in thread "main"
java.lang.NoClassDefFoundError:
xyz

E:\Java>java inherit1
This is class A
This is class B

E:\Java>
*/
```

INHERITANCE BASICS

```
class A
{
private A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit2
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

Private Constructor in super class

```
/*
E:\Java>javac xyz1.java
xyz1.java:12: A() has private access in A
{
^
1 error
*/
```

INHERITANCE BASICS

```
class A
{
private A()
{
System.out.println("This is class A");
}
A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit2
{
public static void main(String args[])
{
B b1 = new B();
} }
```

```
/*
E:\Java>javac xyz2.java
xyz2.java:7: A() is already defined in
A
A()
^
xyz2.java:16: A() has private access
in A
{
^
2 errors
*/
```

When Super class has a parameterized constructor.

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor
of class A");
} }
class B extends A
{
private int b;
private double c;
B(int b,double c)
{
this.b=b;
this.c=c;
System.out.println("This is constructor
of class B");
} }
```

```
B b1 = new B(10,8.6);
```

```
D:\java\bin>javac inhtest.java
inhtest.java:15: cannot find
symbol
symbol  : constructor A()
location: class A
{
^
1 errors
```

INHERITANCE BASICS

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is
constructor of class A");
} }
class B extends A
{
private int b;
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is
constructor of class B");
} }
```

```
B b1 = new B(8,10,8.6);
```

OUTPUT

```
This is constructor of class A
This is constructor of class B
```

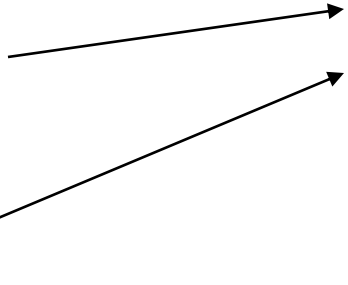
INHERITANCE BASICS

```
class A
{
private int a;
protected String name;
A(int a, String n)
{
this.a = a;
this.name = n;
}
void print()
{
System.out.println("a="+a);
}
}
```

```
class B extends A
{
int b;
double c;
B(int a,String n,int b,double c)
{
super(a,n);
this.b=b;
this.c =c;
}
void show()
{
//System.out.println("a="+a);
print();
System.out.println("name="+name);
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

Can Not use *a*
a is private in
super class

Calls print() from
super class A



INHERITANCE BASICS

```
class xyz3
{
public static void main(String args[])
{
B b1 = new B(10,"OOP",8,10.56);
b1.show();
}
}
```

```
E:\Java>java xyz3
a=10
name=OOP
b=8
c=10.56
```

USE OF super KEYWORD

- Can be used to call super class constrctor

  super();

  super(<parameter-list>);

- Can refer to super class instance variables/Methods

  super.<super class instance variable/Method>

INHERITANCE BASICS

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor
of class A");
}
void print()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display
in A");
}
}
```

```
class B extends A
{
private int b;
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is constructor
of class B");
}
void show()
{
print();
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

INHERITANCE BASICS

```
class inhtest1
{
public static void main(String args[])
{
B b1 = new B(10,8,4.5);
b1.show();
}
}
/* OutPUt
D:\java\bin>java inhtest1
This is constructor of class A
This is constructor of class B
a=10
b=8
c=4.5
*/
```

INHERITANCE BASICS

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor
of class A");
}
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display
in A");
}
}
```

```
class B extends A
{
private int b;
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is constructor
of class B");
}
void show()
{
// show();
super.show();
System.out.println("b="+b);
System.out.println("c="+c);
display();
}
}
```

INHERITANCE BASICS

```
class inhtest1
{
public static void main(String args[])
{
B b1 = new B(10,8,4.5);
b1.show();
}
}
/* OutPut
D:\java\bin>java inhtest1
This is constructor of class A
This is constructor of class B
a=10
b=8
c=4.5
hello This is Display in A
*/
```

INHERITANCE BASICS

```
class A
{
int a;
A( int a)
{ this.a =a; }
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display
in A");
}
}
```

```
class B extends A
{
int b;
double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
}
void show()
{
//super.show();
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

INHERITANCE BASICS

```
class inhtest2
{
public static void main(String args[])
{
B b1 = new B(10,20,8.4);
b1.show();
}
}
/*
D:\java\bin>java inhtest2
a=10
b=20
c=8.4
*/
```

INHERITANCE BASICS

```
class A
{
int a;
A( int a)
{ this.a =a; }
}
```

```
class B extends A
{
int a;   // super class variable a hides here
int b;
double c;
B(int a,int b,double c)
{
super(100);
this.a = a;
this.b=b;
this.c=c;
}
void show()
{
// How can we print the value of super class variable "a"?
System.out.println("Super class a="+super.a);
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
}  }
```

Use of super to refer to super class varible a

INHERITANCE BASICS

```
class inhtest2
{
public static void main(String args[])
{
B b1 = new B(10,20,8.4);
b1.show();
}
}

/* Out Put
D:\java\bin>java inhtest2
Super class a=100
a=10
b=20
c=8.4
*/
```

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding is one of the ways in which Java supports Runtime Polymorphism.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Example

```
class A
{
    void m1()
    {
        System.out.println("Inside A's
m1 method");
    }
}

class B extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside B's m1
method");
    }
}
```

```
class C extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside C's m1
method");
    }
}
```

Example

```
// Driver class
class Dispatch
{
    public static void main(String args[])
    {
        // object of type A
        A a = new A();

        // object of type B
        B b = new B();

        // object of type C
        C c = new C();

        // obtain a reference of type A
        A ref;

        // ref refers to an A object
        ref = a;

        // calling A's version of m1()
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

OUTPUT:

```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

EXAMPLE 2

```
// class A
class A
{
    int x = 10;
}

// class B
class B extends A
{
    int x = 20;
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

Output:

10

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

Advantages of Dynamic Method Dispatch

- Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
- It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Exercise1

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

Output :

The static method in Animal
The instance method in Cat

Exercise1

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Exercise2

Writing Final Classes and Methods

- You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.
- You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm
{……
 final ChessPlayer getFirstPlayer()
            {
            return ChessPlayer.WHITE;
             }
...
}
```

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the **String class.**

OBJECT ORIENTED PROGRAMMING WITH JAVA

Exercise3

In Java, Constructor over-riding is possible?

In Java, Constructor overriding is not possible as the constructors are not inherited as overriding is always happens on child class or subclass but constructor name is same as a class name so constructor overriding is not possible but constructor overloading is possible.

Can we override static method in Java?

No, you cannot override a static method in Java because it's resolved at compile time. In order for overriding to work, a method should be virtual and resolved at runtime because objects are only available at runtime.

Can we overload a static method in Java?

Yes, you can overload a static method in Java. Overloading has nothing to do with runtime but the signature of each method must be different. In Java, to change the method signature, you must change either number of arguments, type of arguments or order of arguments.

Exercise 3: Static vs Dynamic binding

- Static binding is done during compile-time while dynamic binding is done during run-time.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

THANK YOU

Dr. N MEHALA

Department of Computer Science and Engineering

mehala@pes.edu

INTERFACES IN JAVA

1. *Java Does not support Multiple Inheritance directly. Multiple inheritance can be achieved in java by the use of interfaces.*
2. *We need interfaces when we want functionality to be included but does not want to impose implementation.*
3. *Implementation issue is left to the individual classes implementing the interfaces.*
4. *Interfaces can have only **abstract methods** and **final fields**.*
5. *You can declare a variable to be of type interface. But you can not create an object belonging to type interface.*
6. *Interface variable can point to objects of any class implementing the interface.*
7. *Another way of implementing Run Time Polymorphism.*
8. *In an interface, access specifier is by default **public***

Similarities between Interfaces and classes

- is compiled into byte code file
- can be either public,protected, private or package accessibility
- can not be public unless defined in the file having same name as interface name
- serve as a type for declaring variables and parameters

Differences between Interfaces and classes

- Declares only method headers and public constants
- Has no constructors
- Can be implemented by a class
- Can not extend a class
- Can extend several other interfaces

General Form

- ***Syntax :***

<access specifier> interface <interface name> extends [<interface1> , <interface 2> ……]

{

[public][final] variablename 1 = value;

……

[public][final] variablename N = value;

[public][abstract] <return type> methodname 1(<parameter lis>);

[public][abstract] <return type> methodname 2(<parameter lis>);

……

[public][abstract] <return type> methodname N(<parameter lis>);

}

Examples

Should be typed in file A.java

```
public interface  A
{
double PI = 3.14156;
void show();
void display();
}
```

By Default public final Should be initialized

double PI; → Wrong

Can have only abstract methods. Each method is by default public abstract

```
class XYZ implements A
{
public void show()    { ….. }
public void display() { …..  }
}
```

Example 2

```
interface X
{
int x;
void show();
void display();
}
```

```
A.java:3: = expected
int x;
```

Every variable in interface is by defauly public final and hence should be initialized to some value

Implementing Interface Methods

Use public access specifer for implementing interface methods

```
interface X
{
int x =10;
void show();
void display();
}
class A implements X
{
void show()
{
System.out.println("Hello");
}
void display()
{
System.out.println("Hi");
}
}
```

```
E:\oop>javac A.java
A.java:13: display() in A cannot
implement display() in X; attempting
to assign
weaker access privileges; was
public
void display()
     ^
A.java:9: show() in A cannot
implement show() in X; attempting
to assign weaker
access privileges; was public
void show()
     ^
2 errors
```

Use public access specifer for implementing interface methods

```
interface X
{
int x =10;
void show();
void display();
}
class A implements X
{
public void show()
{
System.out.println("Hello");
}
public void display()
{
System.out.println("Hi");
}
}
```

By Default public final

By Default public abstract

- Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
- A variable or method declared without any access control modifier is available to any other class in the same package.
- The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Exercise

1) Can interfaces have constructors?

No. Interfaces can't have constructors. They show 100% abstractness.

2) Can we re-assign a value to a field of interfaces?

No. The fields of interfaces are static and final by default. They are just like constants. You can't change their value once they got.

3) Can we declare an Interface with "abstract" keyword?

Yes, we can declare an interface with "abstract" keyword. But, there is no need to write like that. All interfaces in java are abstract by default.

4) For every Interface in java, .class file will be generated after compilation. True or false?

True. .class file will be generated for every interface after compilation.

5) Can we override an interface method with visibility other than public?

No. While overriding any interface methods, we should use public only. Because, all interface methods are public by default and you should not reduce the visibility while overriding them.

Exercise

6) Can interfaces become local members of the methods?

No. You can't define interfaces as local members of methods like local inner classes. They can be a part of top level class or interface.

7) Can an interface extend a class?

No, a class can not become super interface to any interface. Super interface must be an interface. That means, interfaces don't extend classes but can extend other interfaces.

8) Like classes, does interfaces also extend Object class by default?

No. Interfaces don't extend Object class.

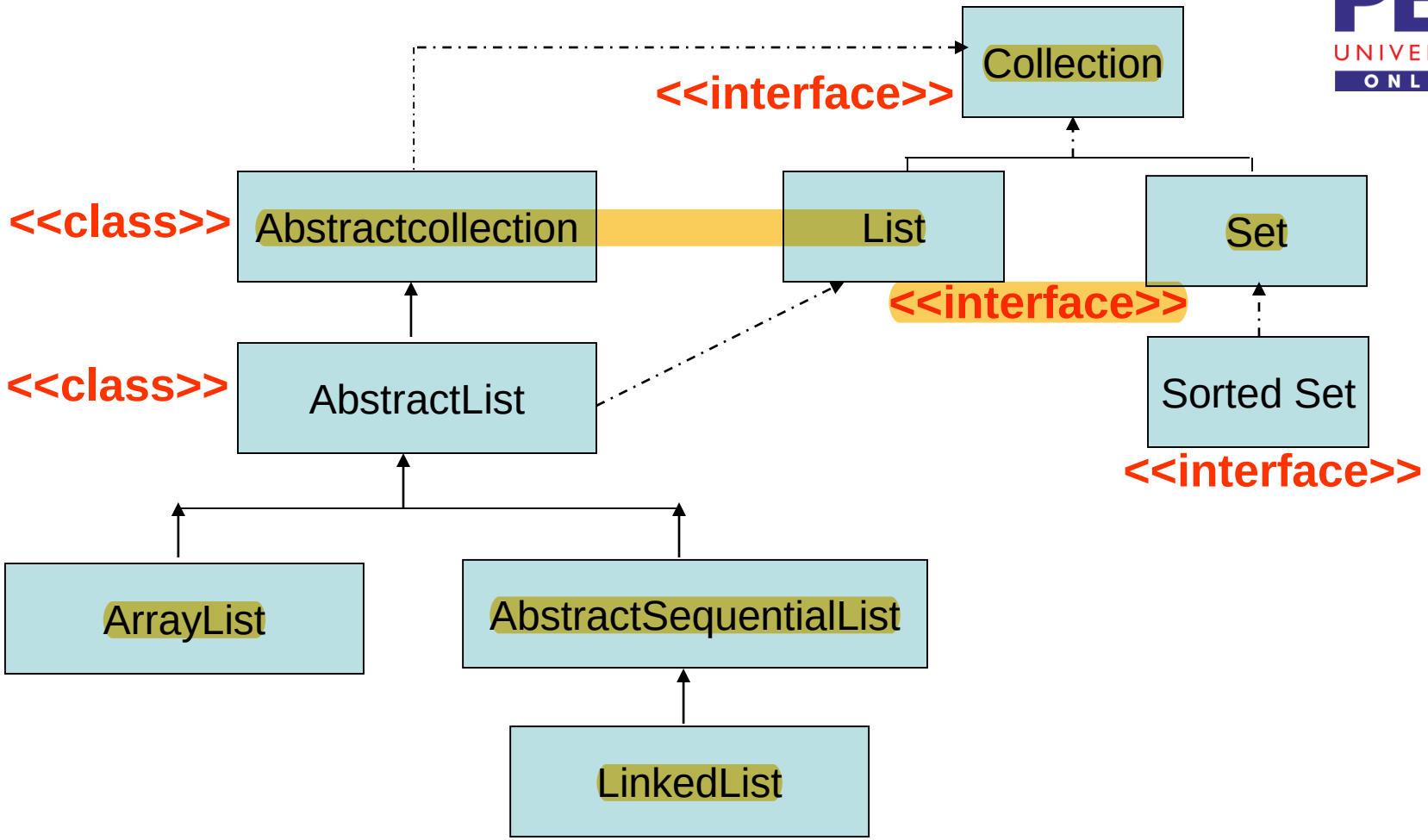
9) Can interfaces have static methods?

No. Interfaces can't have static methods.

http://www.codespaghetti.com/interfaces-interview-questions/

https://www.shristitechlabs.com/java/interviewquestions/top-10-interview-questions-in-interfaces/

Interfaces from Java's Collection Framework



PES UNIVERSITY ONLINE

Collections in Java

Introduction To Java's Collection Framework

What are Collections

- ***Group of Objects treated as a single Object.***
- ***Take group of students and maintain it as a LinkedList. <<Linked List is a Collection>>***
- ***Java provides supports for manipulating collections in the form of***

1. ***Collection Interfaces***
2. ***Collection Classes***

- ***Collection interfaces provide basic functionalities whereas collection classes provides their concrete implementation***

Collection Interfaces

There are Five Collection Interfaces

1. Collection

- ***Enables You to work with collections. << Top of Collection Hierarchy>>***

2. List

- ***Extends Collection to handle list of elements [objects]***
- ***Allows duplicate elements in the list***
- ***Uses indexing technique starting with 0 to access elements***

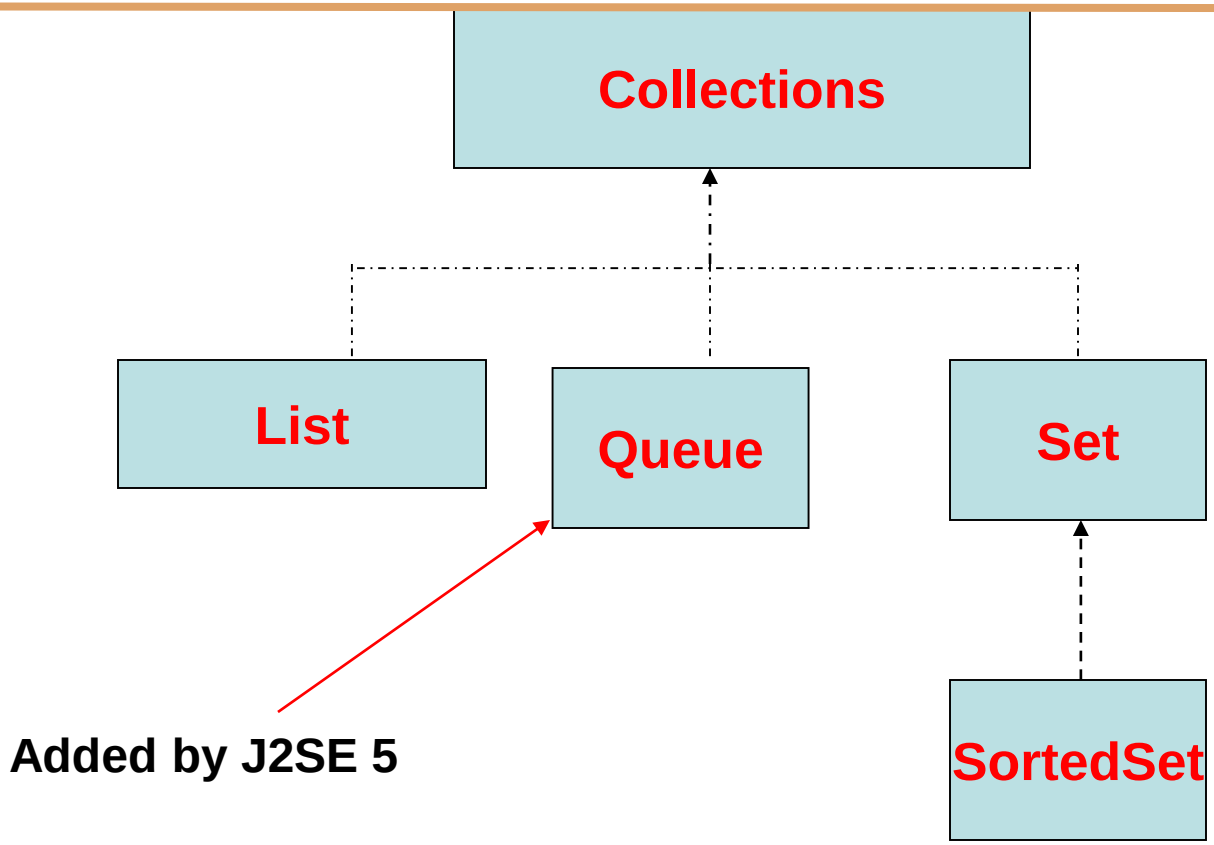
3. Set

- ***Extends Collection to handle set of elements [objects], which must contain unique elements***

4. SortedSet

- ***Extends Set to handle sorted elements in a set***

Collection Interfaces



Collections also uses following interfaces:

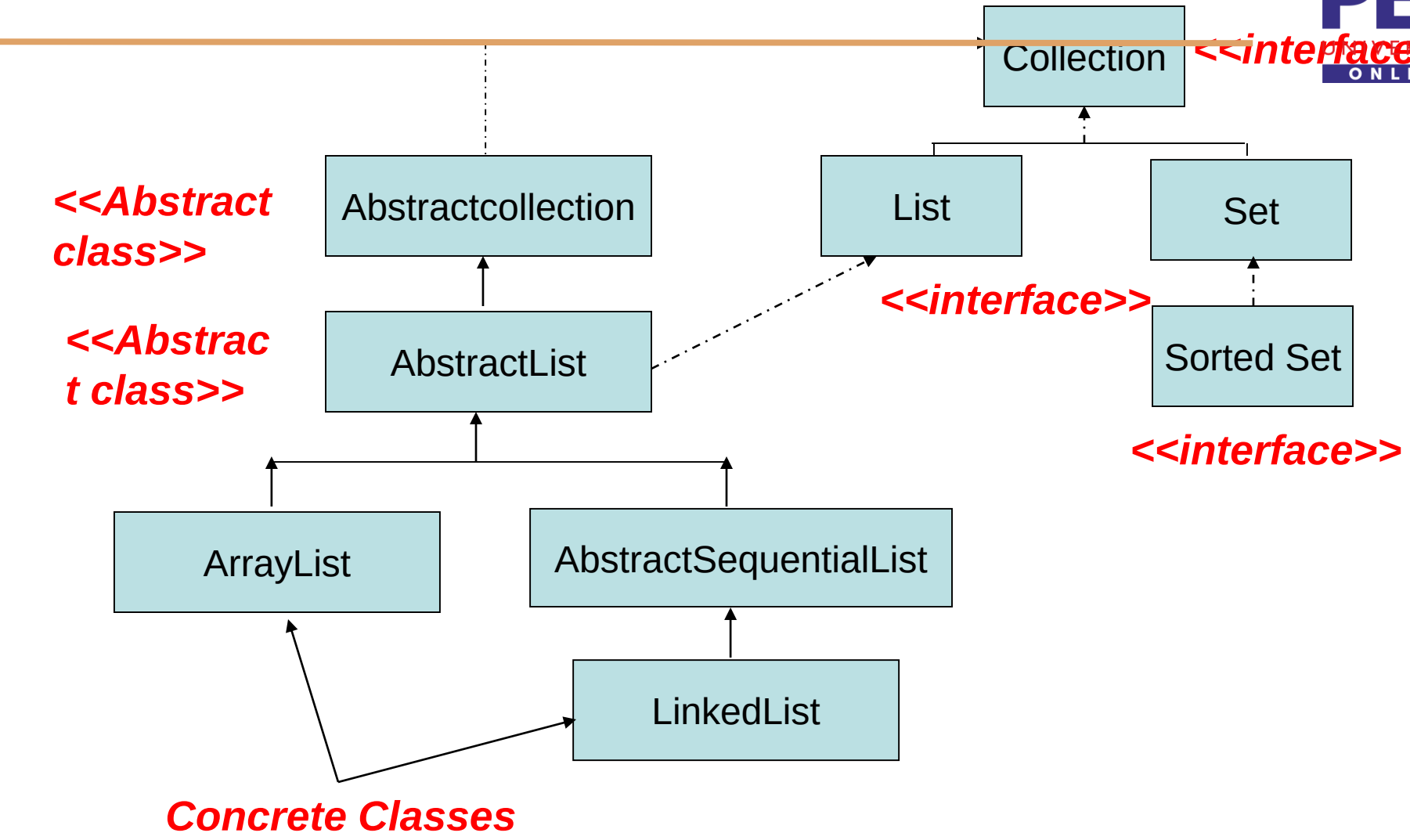
1. Comparator 2. Iterator 3. ListIterator 4. RandomAccess

Collection Classes

- ***Collection classes are standard classes that implement collection interfaces***
- ***Some Collection Classes are abstract and some classes are concrete and can be used as it is.***
- ***Important Collection Classes:***

1. ***AbstractCollection***
2. ***AbstractList***
3. ***AbstractSequentialList***
4. ***LinkedList***
5. ***ArrayList***
6. ***AbstractSet***
7. ***HasSet***
8. ***LinkedHashSet***
9. ***TreeSet***

Partial View of Java's Collection Framework



Important Method in Collection Interfaces

1. boolean add(Object obj) / boolean addAll(Collection c)
   - Adds in to collection either a single Object or all elements from another collection. [Addition only in the end]
2. void clear() // clears all elements from the Collection
3. boolean contains(Object obj)
   - Returns true if obj is there in the collection otherwise false
4. boolean containsAll(Collection c)
   - Returns true if invoking collection contains all elements of c
5. boolean equals(Object obj)
   - Returns true if invoking collection and obj are equal or not
6. boolean isEmpty()
   - Returns true if invoking collection is Empty otherwise false
7. int size() // returns size of collection
8. boolean remove(Object obj) / boolean removeAll(Collection c)
9. Iterator iterator()
   - Returns an iterator for a collection for traversing purpose

Important Method in List Interfaces

1. boolean add(int index, Object obj) / boolean addAll(int index, Collection c)
- Adds in to collection either a single Object or all elements from another collection at a mentioned index.
2. Object get(int index)
- Return object at given index. Index >=0 and < size();
3. int indexOf(Object obj)
- Returns index of obj in the invoking collection otherwise -1
4. int lastIndexOf(Object obj)
- Returns index of obj in the invoking collection from last otherwise -1 will be returned if obj not found
5. ListIterator listIterator()
- Returns a list iterator for a given collection
- ListIterator allows both way traversal . Iterator allows only forward traversal
6. Object remove(int index)
- Removes elements from invoking collection at index. iIndex >=0 and < size();
7. Object set(int index,Object obj)
- Sets the obj as elements for location specified by index. Index >=0 and < size();

ArrayList class

- Supports Dynamic Arrays that can grow as needed.
- Variable length array of object references
- ArrayList can increase or decrease in size.
- Earlier versions of java supports dynamic arrays by a legacy class Vector.

public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

<E> Type of the Objects/Elements stored

Types of ArraysList

1. ***Unparametrized ArrayLists***

- ***Supported in earlier versions of Java (1.3 and earlier)***
- ***Can store/handle objects of any type.***

2. ***Parametrized ArrayLists***

- ***Supported in later versions after 1.4 onwards***
- ***Can handle objects of only mentioned type***

Note :

If you are using unparametrized arraylists and are using latest java compiler then use the following to compile:

```
javac -Xlint <sourcefile>
```

ArrayList Constructors

Unparametrized Type

parametrized Type

1. ***ArrayList()***

- *Empty ArrayList() size() =0*
- *Examples :*

```
ArrayList  arr = new ArrayList();
ArrayList<BOX> boxes = new ArrayList();
ArrayList<Student> students = new ArrayList<Student>();
```

2. ***ArrayList(Collection c)***

- *Creates an ArrayList which is initialized with elements from other collection*

3. ***ArrayList(int capacity)***

- *Creates an arraylist with initial capacity.*
- *Examples*

```
ArrayList  arr = new ArrayList(10);
ArrayList<BOX> boxes = new ArrayList(10);
ArrayList<Student> students = new ArrayList(20);
ArrayList<Student> students1 = new ArrayList<Student>(20);
```

Example Unparametrized ArrayLists

*To Use ArrayList import java.util.**

Empty ArrayList size() == 0, Type is unparametrized

```
import java.util.*;
class list
{
public static void main(String args[])
{
ArrayList arr = new ArrayList();

ArrayList arr1 = new ArrayList(20);

System.out.println(arr.size());
System.out.println(arr.size());

// Adding Elements
```

Unparametrized ArrayList with size() == 0 and capacity = 20

0

Won't work in jdk1.3 and previous versions.

```
// arr.add(10);
arr.add(new Integer(10));          Adds integer 10 at index 0

arr.add("A");                      Adds String "A" at index 1

arr.add(new Double(12.56));        Adds 12.56 at index 2

arr.add(new Boolean(true));        Adds boolean true  at index 3

arr.add(new Integer(30));          Adds integer 30 at index 4

// arr.add(6,new Integer(50)); // IndexOutOfBoundsException

System.out.println(arr.size());  5

arr1.addAll(arr);
}
}
```

Adds all elements of arr to end of arr1

```
E:\oop>javac  list.java
Note: list.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
E:\oop>javac  -Xlint list.java
list.java:13: warning: [unchecked] unchecked call to add(E) as a member of the
raw type java.util.ArrayList
arr.add(new Integer(10));
……….
list.java:21: warning: [unchecked] unchecked call to addAll(java.util.Collection
<? extends E>) as a member of the raw type java.util.ArrayList
arr1.addAll(arr);
          ^
6 warnings
```

```
E:\oop>java list
0
0
5
```

Example Parametrized ArrayLists

*To Use ArrayList import java.util.**

```
import java.util.*;
class list
{
public static void main(String args[])
{
ArrayList<String> arr = new ArrayList();

//ArrayList<BOX> arr1 = new ArrayList(20);

System.out.println(arr.size());
//System.out.println(arr1.size());

// Adding Elements
```

Parametrized ArrayList of type <String>. Can Hold Only String Type Data

Parametrized ArrayList of type <BOX>. Can Hold Only BOX Type Data

Won't work arr can hold only String Data

```
// arr.add(10);
//arr.add(new Integer(10));

arr.add("A");                          Adds String "A" at index 0

//arr.add(new Double(12.56));   Won't Work. arr can hold only String Data

arr.add("B");                          Adds "B"  at index 1

arr.add(new String("OOP"));    Adds "OOP" at index 2

// arr.add(6,new Integer(50)); // IndexOutOfBoundsException

System.out.println(arr.size());  3

//arr1.addAll(arr);
}
}
```

Won't work Elemments of different types

Traversing ArrayLists

- Traversing means visiting thru the arrayList and retrieving individual elements.
- Traversal can be forward or backward
- There can be following ways of traversal

1. Use for(…) loop along
2. Use of Iterator interface [For Forward Traversing Only]
3. Use of ListInterator interface [For Both Way Traversing]

Use for loop

```
import java.util.*;
class arraylis
{
public static void main(String args[])
{
ArrayList<String> arrStr = new ArrayList(20);
arrStr.add("A");
arrStr.add("B");
arrStr.add("X");
arrStr.add("Y");
arrStr.add("Z");
```

```
// For Forward Traversing
System.out.println("Forward");
for(int i=0;i<arrStr.size();i++)
{
String str = arrStr.get(i);
System.out.println("Hello "+str);
}
```

Other form of for loop known as for each in collection

```
/* For Forward Traversing using for each
for(int i : arrStr)
{
String str = arrStr.get(i);
System.out.println("Hello "+str);
}
*/
```

```
// For Backward Traversal
System.out.println("Backward");
for(int i= arrStr.size()-1;i>=0;i--)
{
String str = arrStr.get(i);
System.out.println("Hello "+str);
}
}
}
```

```
E:\oop>java arraylis
Forward
Hello A
Hello B
Hello X
Hello Y
Hello Z
Backward
Hello Z
Hello Y
Hello X
Hello B
Hello A
```

Iterator Interface

- Allows the traversal of collections only in forward direction
- All Collections use iterator interface and provides method for attaching iterator for any collection.

Iterator iterator();

- Methods :

1. ***boolean hasNext()***
   - ***Returns true/false if there exists next element or not***
2. ***E next() / Object next()***
   - ***Returns the next element.***
   - ***Used in conjunction with hasNext()***
3. ***void remove()***
   - ***Removes the element from location pointed to by iterator***

Use of iterator

```
import java.util.*;
class arraylis
{
public static void main(String args[])
{
ArrayList<String> arrStr = new ArrayList(20);
arrStr.add("A");
arrStr.add("B");
arrStr.add("X");
arrStr.add("Y");
arrStr.add("Z");


// How to get an iterator for any collection
Iterator itr = arrStr.iterator();
while(itr.hasNext())
{
String str = itr.next();
System.out.println("Hello "+str);
}
```

```
E:\oop>java arraylis
Forward
Hello A
Hello B
Hello X
Hello Y
Hello Z
```

ListIterator Interface

- Extends Iterator interface
- Allows both way traversal

List*Iterator listIterator();*

- Methods :

1. ***boolean hasNext() / boolean hasPrevious()***
   - ***Returns true/false if there exists next/previous element or not***
2. ***E next() / Object next() || E previous() / Object previous()***
   - ***Returns the next/previous element.***
   - ***Used in conjunction with hasNext()/hasPrevious***
3. ***void remove() / void add(E obj)***
   - ***Removes/adds the element from/to location pointed to by iterator***
4. ***int nextIndex() /int previousIndex()***
   - ***Returns the index of previous/next element index***

Use of Listiterator loop

```
import java.util.*;
class arraylis
{
public static void main(String args[])
{
ArrayList<String> arrStr = new ArrayList(20);
arrStr.add("A");
arrStr.add("B");
arrStr.add("X");
arrStr.add("Y");
arrStr.add("Z");


// How to get an ListIterator for any collection for forward Traversal
System.out.println("Forward");
ListIterator Litr = arrStr.listIterator();
while(Litr.hasNext())
{
String str = (String) Litr.next();
System.out.println("Hello "+str);
}
```

```
// How to get an ListIterator for any collection for Backward Traversal
System.out.println("Backward");
ListIterator Litr1 = arrStr.listIterator(arrStr.size());
while(Litr1.hasPrevious())
{
String str = (String) Litr1.previous();
System.out.println("Hello "+str);
}
}
}
```

```
E:\oop>java arraylis
Forward
Hello A
Hello B
Hello X
Hello Y
Hello Z
```

```
import java.util.*;
class arraylis
{
public static void main(String args[])
{
ArrayList<String> arrStr = new ArrayList(20);
arrStr.add("A");
arrStr.add("B");
arrStr.add("X");
arrStr.add("Y");
arrStr.add("Z");

// For Forward Traversing
System.out.println("Forward");

ListIterator<String> Litr = arrStr.listIterator();
while(Litr.hasNext())
{
String str = Litr.next(); // No Need of type casting
System.out.println("Hello "+str);
}
```

Parametrized Iterator at the start of list

```
// For Backward Traversing
System.out.println("Backward");
ListIterator<String> Litr1 = arrStr.listIterator(arrStr.size());
while(Litr1.hasPrevious())
{
String str = Litr1.previous();
System.out.println("Hello "+str);
}
}
}
```

List Itearator sets at the end of the list

```
E:\oop>java arraylis
Forward
Hello A
Hello B
Hello X
Hello Y
Hello Z
```

Collections.sort()

```
// Java program to demonstrate working of Collections.sort()
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al);

        // Let us print the sorted list
        System.out.println("List after the use of" +
                           " Collection.sort() :\n" + al);
    }
}
```

OUTPUT:

List after the use of Collection.sort() :
[Dear, Friends, Geeks For Geeks, Is, Superb]

Collections.sort()

```
// Java program to demonstrate working of
Collections.sort()
// to descending order.
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al, Collections.reverseOrder());

        // Let us print the sorted list
        System.out.println("List after the use of" +
                           " Collection.sort() :\n" + al);
    }
}
```

Arrays.sort works for arrays which can be of primitive data type also. Collections.sort() works for objects Collections like ArrayList, LinkedList, etc. We can use Collections.sort() to sort an array after creating a ArrayList of given array items.

OUTPUT:

List after the use of Collection.sort() :
[Dear, Friends, Geeks For Geeks, Is, Superb]

Comparable Interface

1. **Provides an interface for comparing any two objects of same class.**

General Form Unparameterized:

```
public interface Comparable
{
int compareTo(Object other );
}
```

General Form Parameterized:

```
public interface Comparable<T>
{
int compareTo(T other );
}
```

<<T>> is the type of object

Note : other parameter should be type caste to the class type implementing Comparable interface for un parametrized Type

Collections. sort method can sort objects of any class that implements comparable interface.

By implementing this interface , programmers can implement the logic for comparing two objects of same class for less than, greater than or equal to.

Examples for Implementation For Unparametrized

```
class BOX Implements Comparable
{
.....
.....
.....
public int compareTo(Object other)
{
BOX box = (BOX) other;
.....Logic for comparison ….
}
.....
}
```

```
class Student Implements Comparable
{
.....
.....
.....
public int compareTo(Object other)
{
Student std = (Student) other;
.....Logic for comparison ….
}
.....
}
```

Example 1 [Importance of comparable]

PES UNIVERSITY ONLINE

// Name of source File comparatorTest.java

```
import java.util.*;
class A
{
int a;
int b;
A(int a,int b)
{
this.a=a;
this.b=b;
}
public String toString()
{
return "a="+a+"b="+b;
}
} // End of class A
```

```
class comparableTest1
{
public static void main(String args[])
{
int a[] ={10,6,8,9,45,-67};
String names[] = {"OOP","Java","UML","list"};
double values[] = {10.56,3.45,8.56,2.67};

Arrays.sort(a);
for(int i=0;i<a.length;i++)
System.out.print(a[i]+" ");
System.out.println("");
Arrays.sort(a, Collections.reverseOrder());
Arrays.sort(names);
for(int i=0;i<names.length;i++)
System.out.print(names[i]+" ");
System.out.println("");
```

*Sorts the Elements of array **a***

Prints the Elements

-67 6 8 9 10 45

*Sorts the Elements of array **names***

Prints the Elements

Java OOP UML list

```
Arrays.sort(values);
for(int i=0;i<values.length;i++)
System.out.print(values[i]+" ");
System.out.println("");
```

Sorts the Elements of array values

Prints the Elements

2.67 3.45 8.56 10.56


```
A[] arr = new A[10];
```

Array of Object References

```
arr[0] = new A(10,6);
arr[1] = new A(8,16);
arr[2] = new A(4,3);
arr[3] = new A(5,21);
arr[4] = new A(34,16);

Arrays.sort(arr);
```

Can not sort elements of arr

```
Arrays.sort(arr);
for(int i=0;i<arr.length;i++)
System.out.print(arr[i]+" ");
System.out.println("");
}
} // End of class comparableTest1
```

TO USE **Arrays.sort()** METHOD FOR OBJECT REFERENCES, THE CLASS OF OBJECT REFERENCES MUST IMPLEMENT **Comparable** OR **Comparator** INTERFACE.

OutPut


```
E:\oop>java comparableTest1
-67 6 8 9 10 45
Java OOP UML list
2.67 3.45 8.56 10.56
Exception in thread "main" java.lang.ClassCastException: A
        at java.util.Arrays.mergeSort(Arrays.java:1156)
        at java.util.Arrays.mergeSort(Arrays.java:1167)
        at java.util.Arrays.sort(Arrays.java:1080)
        at comparableTest1.main(comparableTest.java:46)
```

Example 2 [Importance of comparable]

```
// Name of source File comparatorTest2.java.

// Same Program Using ArrayLists
    import java.util.*;
    class A
    {
    int a;
    int b;
    A(int a,int b)
    {
    this.a=a;
    this.b=b;
    }
    public String toString()
    {
    return "a="+a+"b="+b;
    }
    } // End of class A
```

```
class comparableTest2
{
public static void main(String args[])
{
ArrayList<Integer> arr1 = new ArrayList<Integer>();
ArrayList<String> arr2 = new ArrayList<String>();
ArrayList<Double> arr3 = new ArrayList<Double>();

// Adding into integer arraylist
arr1.add(10);
arr1.add(30);
arr1.add(20);
arr1.add(5);
```

```
// Adding into String arraylist
arr2.add("10");
arr2.add("30");
arr2.add("20");
arr2.add("5");

// Adding into Double arraylist
arr3.add(10.56);
arr3.add(30.12);
arr3.add(20.34);
arr3.add(5.56);

Collections.sort(arr1);
System.out.println(arr1);

Collections.sort(arr2);
System.out.println(arr2);

Collections.sort(arr3);
System.out.println(arr3);
```

```
ArrayList<A> arr4 = new ArrayList<A>();
arr4.add(new A(10,6));
arr4.add(new A(2,4));
arr4.add(new A(5,16));
arr4.add(new A(100,16));

Collections.sort(arr4);
System.out.println(arr4);

}
}
```

```
E:\loop>javac comparableTest2.java
comparableTest2.java:58: cannot find symbol
symbol  : method sort(java.util.ArrayList<A>)
location: class java.util.Collections
 Collections.sort(arr4);
             ^
```

Example 3

```
class BOX implements Comparable
{
private double length;
private double width;
private double height;
BOX(double l,double b,double h)
{
length=l;width=b;height=h;
}
public double getLength() { return length;}
public double getWidth()  { return width;}
public double getHeight() { return height;}
public double getArea()
{
return 2*(length*width + width*height+height*length);
}
public double getVolume()
{
return length*width*height;
}
```

Unparametrized Comparators

```
public int compareTo(Object other)
{
BOX b1 =(BOX) other;
if(this.getVolume() > b1.getVolume())
return 1;
if(this.getVolume() < b1.getVolume())
return -1;
return 0;
}
public String toString()
{
return “Length:”+length+” Width :”+width +” Height :”+height;
}
} // End of BOX class
```

Other parameter has to be type caste to BOX type before use

Sorting Using Arrays

```
import java.util.*;
class ComparableTest
{
public static void main(String[] args)
{
BOX[] box = new BOX[5];
box[0] = new BOX(10,8,6);
box[1] = new BOX(5,10,5);
box[2] = new BOX(8,8,8);
box[3] = new BOX(10,20,30);
box[4] = new BOX(1,2,3);
Arrays.sort(box);
for(int i=0;i<box.length;i++)
System.out.println(box[i]);
}
} // End of class
```

Sorting Using ArrayLists

```
Import java.util.*;
class ComparableTest
{
public static void main(String[] args)
{
ArrayList box = new ArrayList();
box.add( new BOX(10,8,6));
box.add( new BOX(5,10,5));
box.add( new BOX(8,8,8));
box.add( new BOX(10,20,30));
box.add( new BOX(1,2,3));
Collections.sort(box);
Iterator itr = ar.iterator();
while(itr.hasNext())
{
BOX b =(BOX) itr.next();
System.out.println(b);
}
}
}// End of class
```

Problems With Comparable Interface

- ***Method int compareTo(Object obj) needs to be included in the base class itself.***
- ***We can include only single ordering logic.***
- ***Different order requires logic to be included and requires changes in the base class itself.***
- ***Each time we need different order we need to change the code itself.***

Comparison By name

```
import java.util.*;
class Student implements Comparable
{
private String name;
private String idno;
private int age;
private String city;
……..
……..
public int compareTo(Object other)
{
Student std = (Student) other;
return
this.name.compareTo(other.name);
}
public String toString()
{
Return “Name:”+name+”Id
No:”+idno+”Age:”+age;
}
} // End of class
```

```
Student[] students = new
Student[10];
……..
……..
……..
Arrays.sort(students);
for(int i=0 ; i<students.length;i++)
System.out.println(students[i]);
```

OUTPUT List sorted by Name

Comparison By idno

```
import java.util.*;
class Student implements Comparable
{
private String name;
private String idno;
private int age;
private String city;
……..
……..
public int compareTo(Object other)
{
Student std = (Student) other;
return this.idno.compareTo(other.idno);
}
public String toString()
{
Return “Name:”+name+”Id
No:”+idno+”Age:”+age;
}
} // End of class
```

```
Student[] students = new
Student[10];
……
……
……
Arrays.sort(students);
for(int i=0 ; i<students.length;i++)
System.out.println(students[i]);
```

OUTPUT List sorted by idno