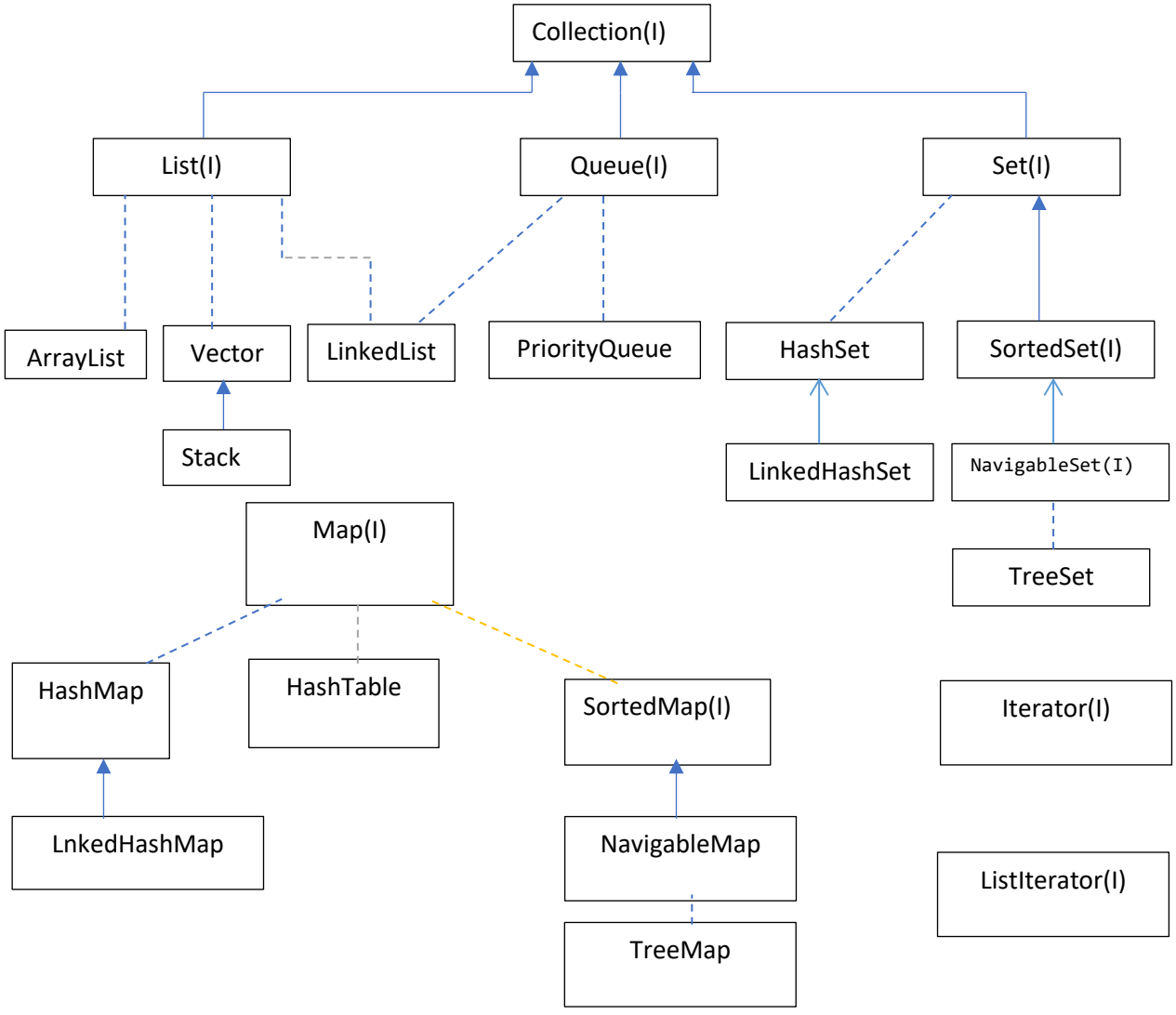
CORE JAVA- PART III

Java collection frameworks: -

- No size limit, grows dynamically
- Add any types of elements
- Store elements of any structure
  i. Linear
  ii. Non-linear
- Provides standard algorithm

What is collection?

- Group of **objects** represented in single entity.
- Objects- same type or different types0
- Objects can be arranged in
  i. Linear
  ii. Non-linear
- Ex. Inbox, Library etc.



CORE JAVA- PART III

Fundamentals of collections: -

1. Add operation
   - Add an elements (objects) into the collection
   - Elements can be any types
   - Elements is casted to Object class type while adding to any collection
   - All elements in collection shows Object class properties
2. Retrieve operation
   - Getting reference to the element present inside the collections
   - Element must be casted to original properties (it shows the object class properties)
3. Remove operation
   - Remove the elements from the collections
   - Removed element still shows object class properties.
   - Down casted to get original properties.

- Collection interface specifies the operation which is common for all types of collections.
- There are 3 types of collections
  i. List
  ii. Que
  iii. Set
- This type of collection provides implementations to the collection interface methods.

1. Public Boolean add(Object e)
   - The methods adds the specified element into the collection
   - If it adds successfully the methods returns true or if the element not added successfully the method returns false

2. Public int size()
   - The methods returns the total number of elements present in the collections
   - If collections are not having any elements then it returns zero.

3. Public boolean isEmpty()
   - The method returns true if the current collection is empty otherwise it returns false.
4. Public void clear()

   - The method removes the all the elements on which the method is called

5. Public boolean contains(Object e)

   - The method is used to find the given elements in the collections
   - If collection contains specified elements then the method returns true.
   - If the collection doesn't contain the specified elements it returns false.

6. Public boolean remove(Object e)
   - The method is used to remove the elements from the collection
   - If the specified element is found it removes it and returns true
   - If the specified elements is not found in the collection it cannot remove it returns false

7. Public Iterator iterator()

- The method on invoking this method the method returns the iterator the object which is used to traverse the elements of collections

8. Public Object[] to Array()
   - On invoking this method it returns the array of objects type containing the elements present in the collection.

Bulk operation method

9. Public boolean addAll(Collection c)
   - The method is used to add all the elements of the given elements into the current collections
   - If the all the elements are added the method returns true otherwise the method returns false.

10. Public boolean containsAll(collection c)
    - The method is used to check whether the current collections contains all the elements of the given collection
    - If it contains returns true otherwise returns false

11. Public boolean removeAll(collection c)
    - The method is used to remove all the elements from the current collection if they are found in the given collection also.
    - Returns true if the elements are removed successfully otherwise false.

12. Public boolean retainAll(Collection c)
    - The method is used to retain all the elements in the current collection which are also found in the given collections
    - Returns true if it retains successfully otherwise false

Lets say c1 in a collection

C1.add(e)

C1.add(e2)

C1.add(e2)

Int n=c1.size();

Sop(n)→ 3

C1. contians(e2)→ true

C1.contains(e1)→ true

C1.isEmpty()→ false

C1.clear()

C1.isEmpty→ true

//lets say c1 is collection

c1.add(e1);

c1.add(e2);

c1.add(e3);

c1.add(e4);

c1.add(e5);

Object objArr=c1.toArray(); -→ collection is converted into array

For(int i=0;i<ObjArr.length;i++)

{

Sop(ObjArr[i]);

}

Lets say c1 is a collection with elements e1, e2,e3

c2 is a another collection with elements

f1,f2,f3,f4

c1.addAll(c2);// add all elements of c2 into c1

c1.size();---→ 7

c1 elements are e1,e2,e3,f1,f2,f3,f4

Lets say c1 is a collection with elements e1, e2,e3

c2 is a another collection with elements

e1,e2,e3,e4

c1.containsAll(c2);// c1 contains all elements of c2

//returns true in this case

Lets say c1 is a collection with elements

e1, e2,e3,f1,f3,e4,e5

c2 is a another collection with elements

f1,f2,f3,f4

c1.removeAll(c2);// remove common elements from c1

c1.size();---→ 5

c1 elements are e1,e2,e3,e4,e5

Lets say c1 is a collection with elements e1, e2,e3

c2 is a another collection with elements

f1,f2,f3,f4

c1.retainAll(c2);// retain common elements of c2 into c1

c1.size();---→ 2

c1 elements are f1, f3

.....

Types of Collections

1. List -- store elements with index
    1) ArrayList
    2) Vector
    3) LinkedList
2. Queue-- store eleents in FIFO
    1) LinkedList
    2) PriorityQueue
3. Set-- store unqiue elements
    1) HashSet
    2) LinkedHashSet
    3) TreeSet
    --
4. Map-- store key value pair
    1) HashMap
    2) Hashtable
    3) LinkedHashMap
    4) TreeMap

Iterator and ListIterator

CORE JAVA- PART III

Collections

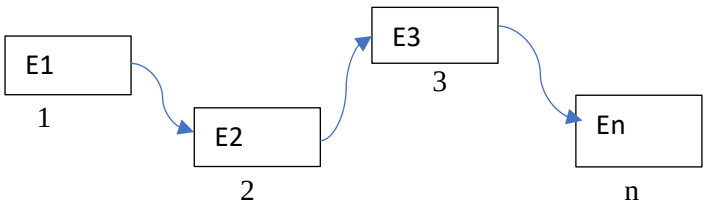
Lists:

- List is a type of collection which stores elements with indexes.
- List is known as index collection because all the elements are stored with index.
- That elements are auto indexed.
- List allows duplicate elements
- List allows null values.
- The list elements can be retrieved or removed based on the index
- On the list, we can perform index based operation like retrieve element based on index, remove elements based on index, insert elements on specified index, replace elements at the specified index.
- The list preserves the order of insertion until and until it is disturbed.
- Lists are two types
  i. Sequential list
  ii. Non-sequential list
- In sequential list, the elements are stored sequentially one after the other
- Where as in non-sequential list the elements are stored randomly in diff location
- Both sequential and non- sequential has indexes.
- The list interface specifies all the operation which can be performed on list type of collections based on indexes.
- It provides only the abstract method for the list operation
- The list interface has 3 implementation classes.
  i. Array list
  ii. Vector
  iii. LinkedList
- The array list and the vector implements the list functionality in a sequential manner
- The linked list class provides implementation to the list functionalities in a non-sequential format.

1. Sequential list - built on array concept

E1	E2	E3	……	En
0	1	2		n

2. Non-sequential list – built on linked list concept



List operations

- Get the elements based on index
- Remove elements based on index
- Insert elements at given index
- Replace element at given index
- Find the index of the specified element.

CORE JAVA- PART III

List interface methods:

1. Public Boolean add(int index, Object e)
   - the method insert the specified element at the specified index of the list
   - returns true if the element is inserted successfully otherwise returns false

2. Public Object get(int index)
   - The method returns reference to the element present at the specified index.
   - If the there is no index present in the list throws exception.
   - Returned reference is of Object class type we can cast it to get original properties.

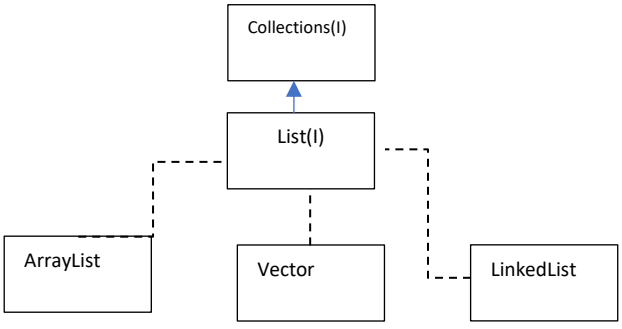
3. Public Object remove (int index)
   - The method removes the object found at the specified index.
   - If the index is not found the method throws exception.

4. Public Object set (int index, Object e)
   - The method is used to replace the elements present at the specified index by the element mentioned in the second argument.
   - It returns the original element which was present before replacing the elements
   - If the index is not found, throws exception.

5. Public int indexof(Object e)
   - The method returns the position of the first occurrence of the specified element from the list
   - If the element is not found it returns -1

6. Public int lastIndexof(Object e)
   - The method returns the last occurrence index of the specified element from the list.
   - If the element is not found returns -1
7. Public List subList(int startIndex, int endIndex)
   - The method returns part of list containing the elements from the specified start index till specified end index, excluding the element at end index.
   - If the indexes are not found the method throws exception

8. Public ListIterator ListIterator()
   - The method returns list iterator type object which is used to iterate the elements of the lists



ArrayList L1=new ArralList();

L1. Add(e1);
L1. Add(e2);
L1. Add(e3);
L1. Add(e2);
L1. Add(e4);
L1. Add(null);

E1	E2	E3	E2	E4	null
0	1	2	3	4	5

L1.add(4,e5)
L1.remove(1);
L1.set(4,e5);

E1	E2	E3	E2	E5	E4	null

E1	E3	E2	E5	E6	null

Java collection Framework: list examples

```
package ListMethods;

import java.util.ArrayList;

public class ListDemo1 {
public static void main(String[] args) {

		ArrayList l1=new ArrayList();
		System.out.println("List size is "+l1.size());

		l1.add("Bheem");
		l1.add("chutki");
		l1.add("kalia");
		l1.add("dolu");
		l1.add("bheem");
		l1.add(null);


		System.out.println("List size is "+l1.size());
		System.out.println("list eelements");
		for (int i = 0; i < l1.size(); i++) {
			System.out.println(l1.get(i));

		}
		l1.add(4,"Bolu");// insert
		l1.set(6, "jaggu");//replace
		l1.remove(5);//remove\\
		System.out.println(".....");
		System.out.println("List size is "+l1.size());
		System.out.println("list eelements");
		for (int i = 0; i < l1.size(); i++) {
			System.out.println(l1.get(i));

}
}
}
```

Output:

```
List size is 0
List size is 6
list eelements
Bheem
chutki
kalia
dolu
bheem
null
.....
List size is 6
list eelements
Bheem
chutki
kalia
dolu
Bolu
jaggu
```

- Array list is a implementation class of list interface.
- ArrayList implements three marker interfaces.
  i. Serializable
  ii. Cloneable
  iii. Random access
- ArrayList is implemented using resizable array data structure.
- ArrayList has a default capacity of 10
- ArrayList grows by using formula
  (current capacity*3/2)+1
- in ArrayList class toString () method is overridden to return ArrayList in comma separated string format
- the Array List has overloaded constructor

ArrayList Constructor:

1. no arg constructor
ex. ArrayList l1=new ArrayList();

- Creates any empty ArrayList Object with capacity 10;

2. Int arg constructor
Ex. ArrayList l1=new ArrayList(n);

- n→ int type
- n represents initial capacity
- creates an empty ArrayList Object with initial capacity of 'n';

3. collection type arg constructor

- if c1→ is collection of 'n' elements'
- ArrayList l1=new ArrayList(c1)
- Creates an empty ArrayList Object with the specified collection c1

ArrayList l1=new ArrayList();

L1.add(e1)
L1.add(e2)
L1.add(e3)
L1.add(e4)
L1.add(e5)
L1.add(e6)
L1.add(e7)
L1.add(e8)
L1.add(e9)
L1.add(e10)

e1	e2	e3	e4	e5	e5	e6	e7	e8	e9

L1.add(e11)

- Capacity = (10*3/2)+1
  -> new array with size 16
  -> now l1 is pointing to new ArrayList

e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11					
0	1	2	3	4	5	6	7	8	9	10					

Vector :

- Vector is a implementation class of list
- It implements three marker interfaces
  i. Serializable
  ii. Cloneable
  iii. Random access
- The vector implementation is same as ArrayList implantation
- The only diff is vector grows by doubling the current capacity.
- Another diff is vector methods are synchronized hence it is known as thread safe class
- Vector is a legacy class, which we won't use in project

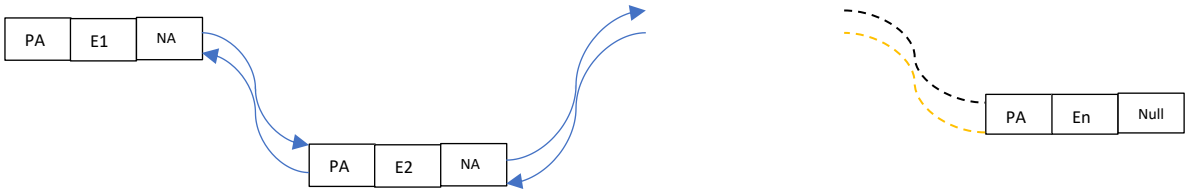
LinkedList:

- LinkedList is a implementation class of list interface
- It also implements queue interface
- LinkedList implements two marker interfaces
  i. Serializable
  ii. Cloneable
- The LinkedList is implemented with doubly linked list data structure
- LinkedList doesn't have default capacity
- LinkedList grows one node at time
- The LinkedList has only one constructor without any arg.

LinkedList l1=new LinkedList()
l1.add(E1);
l1.add(E2);
l1.add(E3);

PA	E3	NA

CORE JAVA- PART III



Where
PA-previous Address
NA-next address

ArrayList:
Adv:
i. Retrial is easy and fast
Dis-adv :
i. Insertion and dilation are slow

LinkedList :
Adv:
i. Insertion and dilation is fast
Dis-adv:
i. Retrieval is slow

```
package ListMethods;

import java.util.ArrayList;
import java.util.List;

public class ListDemo2 {
	public static void main(String[] args) {
		List l1=new ArrayList();


	l1.add(new Student());
	l1.add(new Pen());
	l1.add(new Employee());
	l1.add(new Mobile());
	l1.add(new NoteBook());


//	private void syso("list size = "+l1.size())
		// TODO Auto-generated method stub
for (int j = 0; j < l1.size(); j++) {



		System.out.println(l1.get(j));
	}
	}
```

}

Output :

Without toString overloaded

```
ListMethods.Student@15db9742
ListMethods.Pen@6d06d69c
ListMethods.Employee@7852e922
ListMethods.Mobile@4e25154f
ListMethods.NoteBook@70dea4e

With toString() overloaded
Student []
Pen []
Employee []
Mobile []
NoteBook []
```

```
package ListMethods;

public class Employee {

	int id;
	String name;
	double salary;

	public Employee(int id, String name, double salary) {

		this.id = id;
		this.name = name;
		this.salary = salary;
	}

	@Override
	public String toString() {
		return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "]";
	}


}
```

```
package ListMethods;

public class Mobile {
	String os;
	double price;
	int RamSize;


	public Mobile(String os, double price, int ramSize) {
		super();
		this.os = os;
		this.price = price;
		RamSize = ramSize;
	}
```

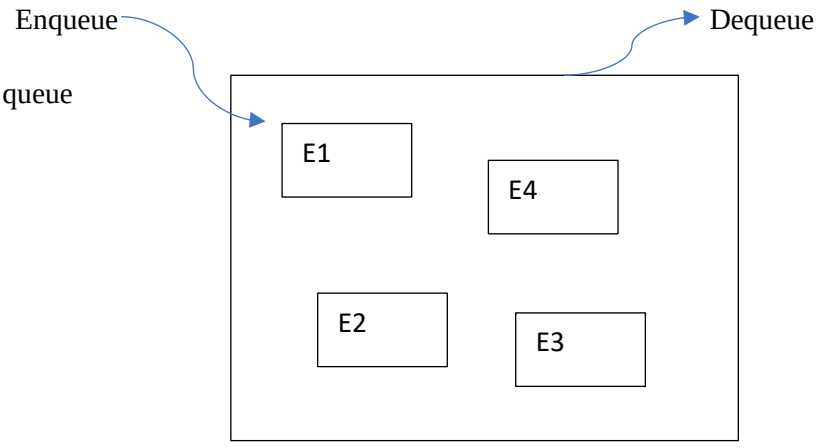
```
	@Override
	public String toString() {
		return "Mobile [os=" + os + ", price=" + price + ", RamSize=" + RamSize + "]";
	}


}
```

```
	package ListMethods;

	import java.util.ArrayList;
	import java.util.List;

	public class ListDemo2 {
			public static void main(String[] args) {
				List<Object> l1=new ArrayList<Object>();


				l1.add(new Student(1234, "hdb", 79.90));

				l1.add(new Employee(123, "gaja", 4000.00));
				l1.add(new Mobile("android", 1000.00, 3));

				l1.add(new Employee(1012, "hdb2", 5000.00));

				l1.add(new Student(146, "mahesh", 89.90));

				System.out.println("Total size : "+l1.size());

				System.out.println("Display each elements.");
				for (int i = 0; i < l1.size(); i++) {
					Object e1=l1.get(i);
					if(e1 instanceof Student) {
						Student s1=(Student)e1;
						System.out.println(s1.rollno+"\t"+s1.name+"\t"+s1.marks);

					}


				}



				System.out.println("Updating garce marks");

				System.out.println("Display each elements.");
				for (int i = 0; i < l1.size(); i++) {
					Object e1=l1.get(i);
					if(e1 instanceof Student) {
						Student s1=(Student)e1;
```

```
                        s1.marks=s1.marks+5;
                    }
                }

                for (int i = 0; i < l1.size(); i++) {
                    Object e1=l1.get(i);
                    if(e1 instanceof Student) {
                        Student s1=(Student)e1;
                        System.out.println(s1.rollno+"\t"+s1.name+"\t"+s1.marks);

                    }
                }
                }
}
```

Queue fundamentals:

- Store elements before processing
- Process elements in FIFO
- Generally not indexed (we can have indexed queue)
- Adding elements into queue is called enqueue.
- Removing elements from queue is called dequeue.
- Doesn't allow null
- Allow duplicate
- A queue has 2 ends
  i. Head or front
  ii. Tail or rear
- Enqueue always happens at tail end of queue.
- Dequeue always happens at head end of queue.



- adding an element into queue
- Ex. Adding E1
- Add 2nd element
- add 3rd element
- Add xth element

Dequeue :

- Remove first element
- Ex. Remove e1 (before head is pointing)

- Remove 2nd element
- Remove 3rd element

.

.

- Remove xth element

Queue:

- ✓ Queue interface defines the operations which can be performed on a queue.
- ✓ This interface is a child interface which extends from a child iterator collection interaface
- ✓ It extends specifications of the collection interfaces
- ✓ In collection frameworks library there are 2 implementations classes which provides implementation to the interface methods
- ✓ The important methods of the queue interface are

1. Public Boolean add(Object e)
   - The method inserts the specified element in the Queue.
   - It queue is a bounded queue if no spaces are available.
   - This method throws exception
2. Public Boolean offer(object e)
   - The method inserts the specified element into the queue. If queue is a bounded queue this will not throw exception
3. Public Object element()
   - Th method retrieves the head element of the queue if we call this method on an empty queue the method throws NoSuchElementException.
4. Public Object peek();
   - The method is used to retrieve the head element from the queue.
   - If we invoke this method on an empty queue it returns null.
5. Public Object remove()
   - The method is used to remove head element from the queue
   - If we call this method in an empty queue the method throws NoSuchElementException
6. Public Object poll()
   - The method remove head element from the queue.\
   - If we call this method on an empty queue it returns null.

Add and offer → enqueue operation
Element() and peek() → examine operation / retrieve operation
Remove() and poll() → Dequeue operation

PriorityQueue Fundamentals:

Enqueue operation	dequeue operation
• Adding 1st element e1	* remove least element of all n elements
• Add 2nd element e2	
• Add 3rd element e3	
• ..	
• Add nth element e_n	

```
package examples;

import java.util.PriorityQueue;
```

CORE JAVA- PART III

```
public class Demo1 {

	public static void main(String[] args) {
		PriorityQueue q1=new PriorityQueue<>();

		q1.add("sangeetha");
		q1.add("poornima");
		q1.add("rekha");
		q1.add("uma");
		q1.add("varsha");
		q1.add("rekha");
		System.out.println(q1.size());
		while(q1.isEmpty()!=true)
		{
			Object e1=q1.poll();
			System.out.println(e1);
		}

		System.out.println(q1.size());
	}

}
```

OutPut:

```
6
poornima
rekha
rekha
sangeetha
uma
varsha
0
```

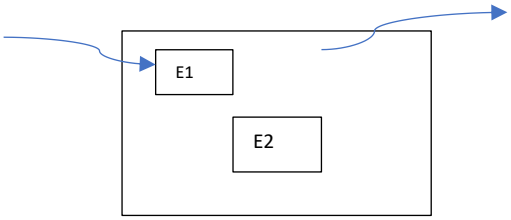
- Queue allows duplicates
- PriorityQueue is an implementation class of Queue interface.
- It implements 2 marker interfaces
- It has overloaded constructors
- It is a unbounded Queue.
- The default capacity of PriorityQueue- 11
- The growing factor is not disclosing
- The PriorityQueue doesn't allow null
- The elements are arranged in natural sorting order.
- Only comparable type of elements are allowed.
- When we remove the element the head element is removed from the queue which is a least element of the queue.

Set:

- Set is a type of collection used to store only enqueue elements
- Set doesn't allow duplicate elements.
- The elements in the set are stored with index randomly.
- Null insertion is allowed.
- Set makes use of HashCode number to decide the object is duplicate or not. If the HashCode are same then Object is limited as duplicate hence it is now allowed in the set.

CORE JAVA- PART III

1. Adding element
   Case 1: adding into set
   Add(e1)
   Case 2: adding e2 into set
   Add(e2)



- Calls equals ()
- Ex. E2.equals(e1)
- If equals returns false then e2 is allowed inside set other wise ignores.
  Case : adding c3 into set
  Add(e3)

Set Interfacing

- It has 3 implementation in child interface of collection
- It specifies set operation
- Interface doesn't have any arg specific method.
- It has 3 implementation classes.
  1. HashSet
  2. LinkedHashSet
  3. TreeSet

  Types of Set:
  There are 3 types of set which are different in implementation
  1. HashSet: - it is a built on HashTable DS, it doesn't preserve order of insertion.
  2. LinkedHashSet: - is a subclass of HashSet, built with hybrid DS, HashTable and LinkedList. It preserves the order of insertion
  3. TreeSet: -
     - it is a type of sorted set
     - The elements are arranged in a natural sorting order.
     - It is built on balance tree DS.

```
package examples;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.TreeSet;

public class Demo2 {
	public static void main(String[] args) {
		HashSet s1=new HashSet<>();
		LinkedHashSet s2=new LinkedHashSet<>();
		TreeSet t1=new TreeSet<>();
		s1.add("hdb");
		s1.add("gaja");
		s1.add("mahesh");
		s1.add("manja");
		s1.add(null);
		System.out.println("set sixe is "+s1.size());
		System.out.println("set eleemnents are.....");
		System.out.println(s1);

	}

}
```

Output:

```
set sixe is 5
set eleemnents are.....
        [null, gaja, mahesh, hdb, manja]
```

HashSet:

- HashSet set is a implementation class of set interface.
- It implements two marker interfaces
  i. Serializable
  ii. Cloneable
- It is implemented by using HashTable(originally HashMap)
- The default capacity of HashSet is 16.
- It grows based on the load factor.
- The default load factor is 0.75.

Constructors of HashSet:

- HashSet has overloaded constructor
  i. No arg constructor
     Ex. HashSet s1=new HashSet ();
     - Creates an empty HashSet with a default capacity of 16.
     - The load factor is 0.05
  ii. Int arg constructor
     - HashSet s1=new HashSet(n)
     - Creates an empty HashSet with the specified initial capacity and with default load factor
  iii. Two arg constructor
     - First arg → int type → to decide initial capacity
     - Second arg → float type→ to decide load factor
     - HashSet s1=new HashSet (n, m)
     - Creates an empty HashSet with the specified initial capacity and specified load factor
  iv. Collection type arguments constructor
     - HashSet s1=new HashSet(c1);
     - C1- collection type argument
     - Creates an HashSet with the elements of specified collection.

How HashSet or LinkedList grows?

- ✓ Based on load factor (fill ratio)
  - Ex. C1 is a capacity of set
  - lf is a load factor

- Set grows when set size value reaches lf % of c1
- C=16 then lf=75% of 16=12

✓ The HashSet or LinkedHashSet stats growing based on the load factor
✓ when the size of a set reaches load factor % of current capacity then the set start growing
✓ Ex. If the current size capacity =16 and load factor =0.75 (75%)
✓ The set will grow when the size reaches 12 i.e. 75% of 16.

Key	Values
#code	E1
#code	E2
#code	E3
#code	E12

```
HashSet s1=new HashSet()\

S1.add(E1)

S1.add(E2)

..

S1.add(E12)
```

LinkedHashSet :-

✓ It is a subclass of HashSet implemented with LinkedList and HashTable data structure.
✓ The LinkedHashSet has some features of HashSet, same method, and same constructor
✓ The only difference is the LinkedHashSet preserve the order of iteration because each entries in the HashTable are Linked.

TreeSet:

✓ TreeSet is a implementation class of navigable set.
✓ The navigable set is a subset of sorted set
✓ Here TreeSet are sorted set and navigable.
✓ The TreeSet class implements two marker interface
  i. Serializable
  ii. Cloneable

✓ The TreeSet is built on Tree Data Structure grows one node at a time.
✓ It doesn't have default capacity
✓ Null insertion is not allowed in the TreeSet
✓ TreeSet allows to insert only the object which implements comparable interface otherwise the TreeSet throws ClassCastException
✓ The elements in the TreeSet are stored by default in a natural sorting order.

Constructor of TreeSet:

✓ The TreeSet has overloaded constructor

1. No arg constructor
   - TreeSet t1=new TreeSet()
   - Creates an empty TreeSet where the elements are sorted normally based on the implementation of comparable interface
2. Comparator type arg constructor:
   - TreeSet t1=new TreeSet(c1)
   - Creates an empty TreeSet, where the elements are sorted based on the implementation of comparator interface.

CORE JAVA- PART III

Cursor:

✓ It is used to traverse (one by one) each element of collection

Types of cursor:

1. Enumeration → unidirectional cursor, it is legacy, not in use
2. Iterator→ unidirectional cursor, widely used and can be used on any type of collections
3. ListIterator→ Bidirectional cursor and It can be used on List

In JCF Library :

- Iterator and ListIterator are interfaces
- Library has not exposed the implementation of these interfaces

Iterator interfaces methods:

1. Public Object Next()
   - Moves the cursor to next element of collection, returns the reference to that element
   - If next element is not present it throws NoSuchElementException
2. Public Object remove()
   - Remove the element where cursor is present.
3. Public Boolean hasNext()
   - Returns true if the next element is present in the collection otherwise returns false.

ListIterator Interface methods:

1. Public Object Next()
   - Moves the cursor to next element of collection, returns the reference to that element
   - If next element is not present it throws NoSuchElementException
2. Public Object remove()
   - Remove the element where cursor is present .
3. Public Boolean hasNext()
   - Returns true if the next element is present in the collection otherwise returns false
4. Public Object previous()
   - Moves the cursor to previous element of collection, returns the reference to that element
   - If previous element is not present it throws NoSuchElementException
5. Public Object hasPrevious()
   - Returns true if the previous element is present in the collection otherwise returns false

How to use Iterator :

ArrayList l1=new ArrayList();\

e1	e2	e3	e4	e5

l1.add(e1);
l1.add(e2);
l1.add(e3);
l1.add(e4);
l1.add(e5);

```
Iterator i1=l1.iterator();
Object o1=l1.next();
Object o2=l1.next();
Object o3=l1.next();
Object o4=l1.next();
Object o5=l1.next();
```

OR

```
While(l1.hasNext())
{
Object o1=l1.next();
}
```

ListIterator:

```
ListIterator i1=l1.ListIterator();
```

```
While(l1.hasNext())
{
Object o1=l1.next();
}
```

And

```
While(l1.hasPrevious())
{
Object o1=l1.previous ();
}
```

- The Collection Framework Library provides cursors to Iterate the elements of collections.
- It provides a common way of accessing collection elements
- There are three types of cursors
  i. Enumeration
  ii. Iterator
  iii. List iterator
- If we want to have a common approach to iterate elements of any collections **irrespective of its implementation we use iterator.**

Program to print student details based on unique RollNo.

```
package SetExamples;

public class Student {
	int rollno;
	String name;
	double marks;
	public Student(int rollno, String name, double marks) {
```

```
		super();
		this.rollno = rollno;
		this.name = name;
		this.marks = marks;



	}


 @Override
//treat RollNo as a unique identification
	public int hashCode() {

		return rollno;
	}
	@Override
	public boolean equals(Object obj) {

		return this.hashCode()==obj.hashCode();
	}
	@Override
	public String toString() {
		return "Student [rollno=" + rollno + ", name=" + name + ", marks=" + marks + "]";
	}


}



package SetExamples;

import java.util.HashSet;
import java.util.Iterator;

public class Demo2 {

	public static void main(String[] args) {
		HashSet s1=new HashSet<>();

		s1.add(new Student(121, "hdb1", 99.99));
		s1.add(new Student(122, "hdb2", 98.99));
		s1.add(new Student(123, "hdb2", 98.99));

		s1.add(new Student(124, "hdb3", 98.99));
		s1.add(new Student(124, "hdb4", 90.99));



		Iterator i1=s1.iterator();
		System.out.println("RollNo\tName\tMarks");
		System.out.println(".................");
		while(i1.hasNext())
		{
			Student s11=(Student)i1.next();
			System.out.println(s11.rollno+"\t"+s11.name+"\t"+s11.marks);
		}

	}
```

}

Output:

```
RollNo  Name    Marks
.................
121     hdb1    99.99
122     hdb2    98.99
123     hdb2    98.99
124     hdb3    98.99
```

Sorting : -

- ✓ Array elements in ascending or descending order.
- ✓ Elements are compared first and then sorted.
- ✓ To compare elements both elements must have same property
  - i. Same type of elements are easy to compare
  - ii. Different type elements can be compared if they have at least one common property
- ✓ Comparison is done either using
  - a. Relational operator
  - b. Subtraction method
- ✓ For ex. 10, 8, 2, 12

  1st 2nd

  Ascending order:
  1. Relational
     - 1st element <2nd element
     - If true → No swap
     - If yes → swap
  2. Subtraction
     - 1st element -2nd element
     - If +ve → swap else No swap

Comparison of class Employee:

```
package SetExamples;

public class Employee implements Comparable{
        int id;
        String name;
        double salary;
        public Employee(int id, String name, double salary) {
                super();
                this.id = id;
                this.name = name;
                this.salary = salary;
        }
        @Override
        public String toString() {
```

```
		return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "]";
	}
	@Override
	public int compareTo(Object o) {
		Employee e1=(Employee)o;

		return this.id-e1.id;

//		return (int) (this.salary-e1.salary);
//		return this.name.compareTo(name);

	}


}


package SetExamples;

import java.util.Iterator;
import java.util.TreeSet;

public class Demo3 {
	public static void main(String[] args) {
		TreeSet s1=new TreeSet<>();

		s1.add(new Employee(1234, "hdb", 1500.00));
		s1.add(new Employee(1335, "hdb2", 1500.00));

		s1.add(new Employee(1346, "hdb2", 2500.00));

		s1.add(new Employee(1134, "hdb3", 75000.00));

		Iterator i1=s1.iterator();
		System.out.println("id\tName\tsalary");
		System.out.println(".................");
		while(i1.hasNext())
		{
			Employee s11=(Employee)i1.next();
			System.out.println(s11.id+"\t"+s11.name+"\t"+s11.salary);
		}


	}

}
```

Output:

```
id	Name	salary
.................
1134	hdb3	75000.0
1234	hdb	1500.0
1335	hdb2	1500.0
1346	hdb2	2500.0
```

CORE JAVA- PART III

Comparable :

- In collection framework library two implementation classes sorts the elements added or stored.
- Ex. PriorityQueue and TreeSet
- Since the elements is an object and object has multiples properties to sort the objects we should specify the properties on which it must be sorted.
- Hence these two classes allows only comparable type object.
- Any class which implements the comparable interface is known as comparable type.
- The comparable interface is defined in java.lang. packages. it has one abstract method **public int** compareTo(Object o) ;
- The class which implements comparable interface should provide the implementation to the compareTo method.
- The implementation method should specify on which property the object must be compared.
- While implementing this method the comparison should be done by taking the difference.
- Whenever we have the comparable type object in TreeSet or PriorityQueue, both the classes arrange the elements in natural sorting order interface.
- We can change the implementation to get in a descending order.
- The comparable interface allows to compare only on single property of a object .
- If we have to compare multiple properties based on the requirement or need we go for comparator interface.

Comparator Interface:

- This interface specifies to compare objects on multiples properties
- The interface is define in java.util. package.
- This interface has two methods
  i. public int equals(Object o)
  ii. public int compare(Object o1, Object o2)
- **the class which implements comparator interface is known as comparator type.**
- **The class should provide implementation to both the methods**
- **The compare method implementation is used to sorting the object whereas the equals method is used to in checking the equality b/w the objects**
- **We can define multiple classes which implements comparable interface.**

Note :

- **If sample one is implementation class of comparator interface m it is not compulsory for class to give implementation to equals method but sample class give implementation to compare method**

Reason:

- **The sample1 inherits concrete equals method for object class since the method declaration are same no need to provide additional implementation.**

Program to arrange the employee details by ID wise, name wise and salary wise

```
package SetExamples;

import java.util.Comparator;

public class BySalaryComparator implements Comparator{

	@Override
	public int compare(Object o1, Object o2) {
		Employee e1=(Employee)o1;
		Employee e2=(Employee)o2;
		return (int) (e1.salary-e2.salary);
	}


}
```

```
package SetExamples;

import java.util.Comparator;

public class ByNameComparator implements Comparator {

	@Override
	public int compare(Object o1, Object o2) {
		Employee e1=(Employee)o1;
				Employee e2=(Employee)o2;

		return e1.name.compareTo(e2.name);
	}



}
```

```
package SetExamples;

import java.util.Iterator;
import java.util.Scanner;
import java.util.Set;
import java.util.TreeSet;

public class Demo3 {


	static void displayEmployeeDetails(Set arg)
	{
		Iterator i1=arg.iterator();
		System.out.println("id\tName\tsalary");
		System.out.println(".....");
		while(i1.hasNext())
		{
			Employee s11=(Employee)i1.next();
			System.out.println(s11.id+"\t"+s11.name+"\t"+s11.salary);
		}
		System.out.println(".....");
```

```
	}
	public static void main(String[] args) {

		Scanner scan=new Scanner(System.in);
		TreeSet s1=new TreeSet<>();

		s1.add(new Employee(1234, "hdb1", 1500.00));
		s1.add(new Employee(1335, "hdb2", 1600.00));

		s1.add(new Employee(1346, "hdb3", 2500.00));

		s1.add(new Employee(1134, "hdb4", 75000.00));

	displayEmployeeDetails(s1);

		System.out.println("Enter 1 to arrange by Employee Name\n\t 2 to arrange by Employee
salary");

	int input=scan.nextInt();
	if(input==1)
	{
		TreeSet s2=new TreeSet<>(new ByNameComparator());
		s2.addAll(s1);
		displayEmployeeDetails(s2);
	}else if(input==2)
	{
		TreeSet s2=new TreeSet<>(new BySalaryComparator());
		s2.addAll(s1);
		displayEmployeeDetails(s2);
	}


	}

}
```

Output:

```
Run 1:
id	Name	salary
.................
1134	hdb4	75000.0
1234	hdb1	1500.0
1335	hdb2	1600.0
1346	hdb3	2500.0
.................
Enter 1 to arrange by Employee Name
	 2 to arrange by Employee salary
2
id	Name	salary
.................
1234	hdb1	1500.0
1335	hdb2	1600.0
1346	hdb3	2500.0
1134	hdb4	75000.0
.................
Run 2:
id	Name	salary
.................
1134	hdb4	75000.0
1234	hdb1	1500.0
1335	hdb2	1600.0
```

```
1346    hdb3    2500.0
.................
Enter 1 to arrange by Employee Name
        2 to arrange by Employee salary
1
id      Name    salary
.................
1234    hdb1    1500.0
1335    hdb2    1600.0
1346    hdb3    2500.0
1134    hdb4    75000.0
.................
```

An employee Object: Arranged in ascending order: By Id property	DisplayEmployeeDetails: Id: int Name: String Salary: double

1. List --?
2. Queue – PriorityQueue
3. Set – TreeSet

Collections:

- Overloaded sort method
- Methods are static

1. Public static void sort (List arg)
   - Sort the elements of List mutually
2. Public static void sort (List arg1, comparator arg2)
   - Sort the elements of List, Not mutually

Java program to arrange Employee details by Their ID , Name , Salary using Collections methods

```
package SetExamples;

import static java.util.Collections.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;


public class Demo4 {


        static void displayEmployeeDetails(List arg)
        {
```

```
		Iterator i1=arg.iterator();
		System.out.println("id\tName\tsalary");
		System.out.println(".....");
		while(i1.hasNext())
		{
			Employee s11=(Employee)i1.next();
			System.out.println(s11.id+"\t"+s11.name+"\t"+s11.salary);
		}
		System.out.println(".....");
	}

	public static void main(String[] args) {

		List l1=new ArrayList();

		ByNameComparator namecomp=new ByNameComparator();
		BySalaryComparator salcomp=new BySalaryComparator();
		l1.add(new Employee(1234, "hdb1", 1500.00));
		l1.add(new Employee(1335, "hdb2", 1600.00));

		l1.add(new Employee(1346, "hdb3", 2500.00));

		l1.add(new Employee(1134, "hdb4", 75000.00));
		displayEmployeeDetails(l1);

		System.out.println("Sorted List Id wise");
		sort(l1);
		displayEmployeeDetails(l1);

		System.out.println("Sorted List - Salary wise");
		sort(l1,namecomp);
		displayEmployeeDetails(l1);

		System.out.println("Sorted List - Name wise");
		sort(l1,namecomp);
		displayEmployeeDetails(l1);

	}

}
```

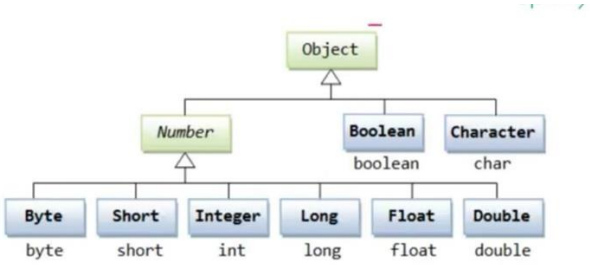
Output:

```
id	Name	salary
.....
1234	hdb1	1500.0
1335	hdb2	1600.0
1346	hdb3	2500.0
1134	hdb4	75000.0
.....
Sorted List Id wise
id	Name	salary
.....
1134	hdb4	75000.0
1234	hdb1	1500.0
1335	hdb2	1600.0
1346	hdb3	2500.0
.....
Sorted List - Salary wise
id	Name	salary
.....
```

```
1234    hdb1    1500.0
1335    hdb2    1600.0
1346    hdb3    2500.0
1134    hdb4    75000.0
.................
Sorted List - Name wise
id      Name    salary
.................
1234    hdb1    1500.0
1335    hdb2    1600.0
1346    hdb3    2500.0
1134    hdb4    75000.0
.................
```

Wrapper Classes in java:

- Java is not a pure object oriented.
- Any language which makes use of primitive data types of data such languages are not object oriented.
- Ex. Pure object oriented languages → Scala, Smalltalk, Eiffel etc.
- The primitive data type are not objects, they do not belong to any class
- Sometimes, it is required to convert data types into objects in java language.
- A data can be converted into an object and then used with the help of wrapper class.



- Each of Java's 8 primitive data types has a dedicated class. These classes are known as Wrapper classes
- Because they wrap the primitive types into an object of that class.
- The wrapper classes are part of java.lang packages which is imported default into all java programs
- the wrapper classes in java serve two primary purposes
  - i. to provide mechanism to wrap primitive value in an object so that primitive can do activities reserved for the objects like being added to ArrayList , HashList
  - ii. To provide utility functions for primitive types converting primitive types to/from string object and also converting into various bases like binary, octal, hex decimal or comparing various object.
  - ➢ To wrap each primitive data types, there comes a Wrapper class.
  - ➢ Eight Wrapper classes exist in java.lang. packages

Primitive Data types	Wrapper classes
byte	Byte

CORE JAVA- PART III

float	Float
int	Integer
double	Double
char	Character
short	Short
long	Long
boolean	Boolean

Boxing and Un-boxing:

- Converting of primitive data type to object type is known as Boxing
- Ex.
  Int k=100;
  Integer i1=new Integer(k);
  double d1=3.3;
  Double d2=new Double(d);

- Converting of object data type to primitive type is known as Un-Boxing.
- Ex.
  Int m=i1.intValue();
  System.out.println(m);//100;
  Double n=d2.doubleValue();
  System.out.println(n);//3.3

- All Wrapper classes are final classes
- All wrapper classes implements Comparable type.
- All wrapper classes overrides 3 object class methods
  toString ()
  hashCode ()
  equals (Object obj)

Wrapper classes in collection Frameworks:

- Wrapper classes are used in collection frameworks.
- In collection we can add only object type of data
- If primitive data added in collection frameworks then auto boxing and auto un_boxing both happens implicitly.
- Ex. ArrayList a1=new ArrayList();
  a1.add(12)
  a1.add(4.5)
- Here, first auto boxing happens and then up-casting to object class happens

Parsing :

- Converting of String type of data to Number format is called parsing
- For parsing we make use of parseX() methods which are present in Wrapper class
- Ex. String str=”123”;
  Int i=Integer.parseInt(str);
  System.out.println(i);//123;
- Parsing has to be done carefully else we may get exception

- In previous example. If String value is “123g” I this case during parsing the value doesn’t match with Integer we get “NumberFormatException”

Map:

- Map is a part of collection frameworks, which stores the elements in key-value pair.
- The value is mapped to the key and stored in the map.
- Key can be any type of object and value can be any type of object
- The key must be unique.
- The value can be duplicate.
- Key is a index to fetch the value from a map.
- While retrieving or removing the value, we must be specify the corresponding key

Map Interfacing :

✓ The map interface specifies the set of operation that can be performed on a any type of map
✓ The interface is implemented in 3 implementation class
  - hashMap
  - HashTable
  - TreeMap

✓ Linked HashMap is a type of hashMap which is a subclass of hasMap.

Map interface methods:

1. Public Boolean put(Object key,Object Value)
   ✓ The method associates the specified value with the specifies key and store the value
   ✓ Returns true if it is successful otherwise false
   ✓ If already key is present in the map it replaces the value of that particular key. Otherwise it acts as new key value pair.
2. Public Object get(Object Key)
   ✓ The method searches for the specified key in the map if found returns the value associated to the key.
   ✓ If key is not found the method returns null
3. Public Object remove(Object key)
   ✓ The method looks for the specified key in the map
   ✓ If key is found it removes the entire pair returning the value mapped to the key.
   ✓ If key is not found, returns null.
4. Public Boolean containskey(Object Key)
   ✓ The method is used to search a key In the map it looks for the specified key in the map if found returns true if not returns false.
5. Public Boolean containsValues(Object Value)
   ✓ The method is used to find the value in the map
   ✓ If the specified is found it returns true if not found returns false.
6. Public Set keyset()
   ✓ The method returns the set of all keys present in the map
   ✓ The return type is a set, we can iterate the elements of the set using iterator.
7. Public collection Values()
   ✓ The function returns the collection of value present in the map.
   ✓ The return type is a collection type.
8. Public int Size()
9. Public void clear()
10. Public Boolean isEmpty()

CORE JAVA- PART III

Program to illustrate the map examples

```
package mapexamples;

import java.util.HashMap;

public class Demo1 {

	public static void main(String[] args) {

		HashMap m1=new HashMap<>();
		System.out.println("Map size is "+m1.size());
		m1.put(12, "java");
		m1.put(54, "j2ee");
		m1.put(34, "andriod");
		m1.put(11, "Google");

		System.out.println(m1);
		System.out.println("Map size is "+m1.size());

		System.out.println("adding duplicTe key ");
		m1.put(11, "Yahoo");
		System.out.println(m1);

		System.out.println("accessing 11 value is  "+m1.get(11));

		m1.remove(54);
		System.out.println("map size after removing one element =  "+m1.size());
	}

}
```

Output :

```
Map size is 0
{34=andriod, 54=j2ee, 11=Google, 12=java}
Map size is 4
adding duplicTe key
{34=andriod, 54=j2ee, 11=Yahoo, 12=java}
accessing 11 value is  Yahoo
map size after removing one element =  3
```

program to traverse the map elements

```
package mapexamples;

import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.TreeMap;

public class Demo2 {
public static void main(String[] args) {
		HashMap m1=new HashMap<>();
//		LinkedHashMap m1=new LinkedHashMap();
//		TreeMap m1=new TreeMap<>();
		System.out.println("Map size is "+m1.size());
```

```
        m1.put(12, "java");
        m1.put(54, "j2ee");
        m1.put(67, "andriod");
        m1.put(11, "Google");
        System.out.println("after adding, Map size is "+m1.size());
        Set s1=m1.keySet();
        Iterator i1=s1.iterator();

        System.out.println("key\t"+"Value ");
        while(i1.hasNext())
        {
                Object key=i1.next();
                Object value=m1.get(key);

                System.out.println(key+"\t"+value);

        }

}
}
```

Output:

```
Map size is 0
after adding, Map size is 4
key     Value
67      andriod
54      j2ee
11      Google
12      java
```

Notes:

- The collections are by default type unsafety
- We can keep any type of object in the collections
- If we want to achieve type safety, we should go for generics while dealing with collections.

Type Unsafety:

```
ArrayList l1=new ArrayList();
l1.add("hdb");
l1.add(new Student(123, "hdb", 76.77));
l1.add(12);
```

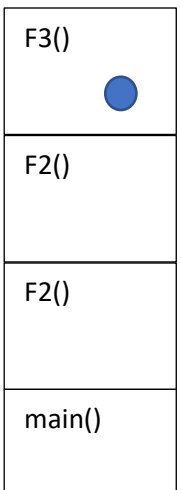
Type Safety :

```
ArrayList<Student> stlist=new ArrayList<Student>();
stlist.add(new Student(123, "hdb", 87.99));
```

Exceptions:

- Function execution/method execution
- LIFO

Stack Elements



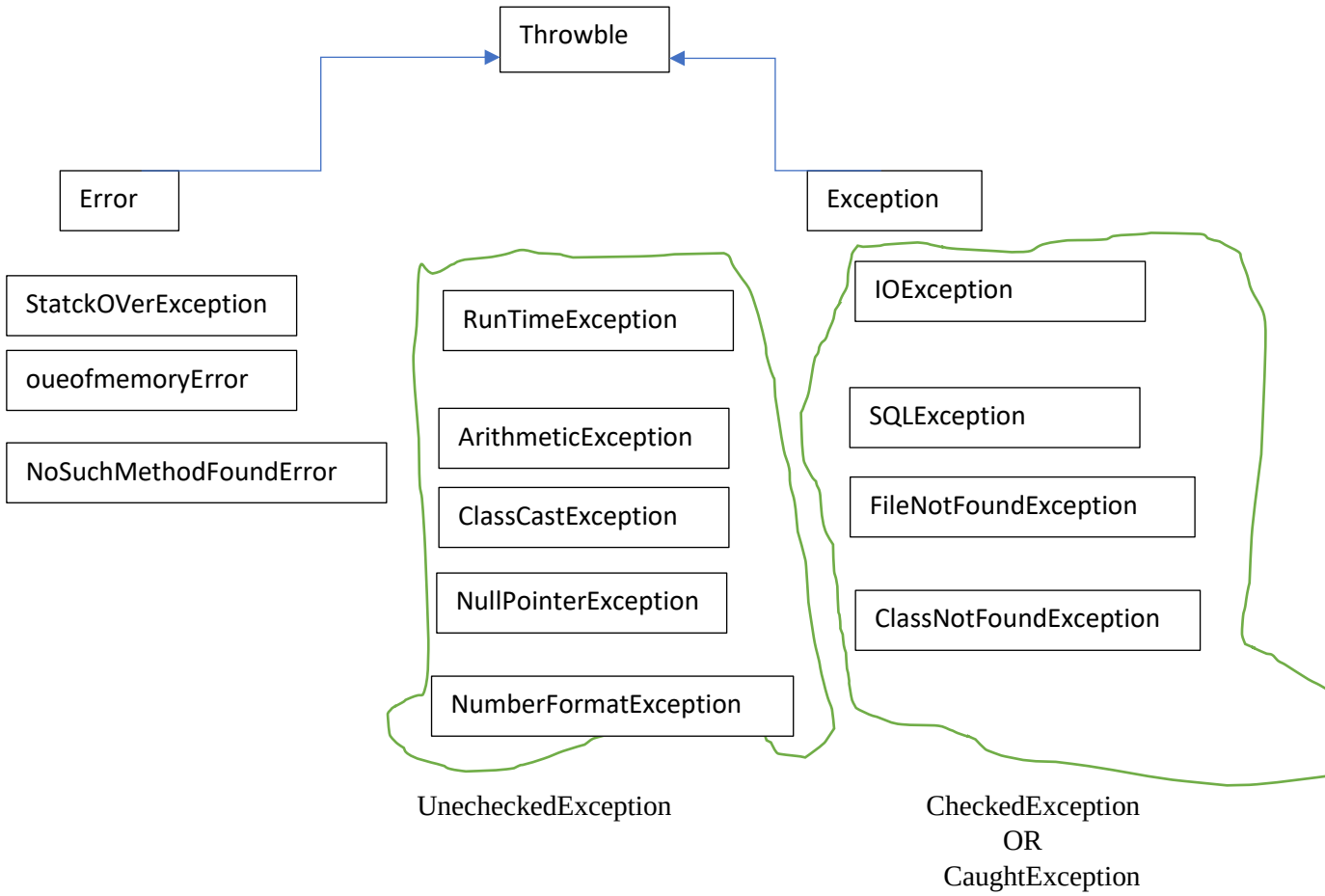
RunTime system
Raise of Exception

Exception Handler → Block of code to the exception

CORE JAVA- PART III

- An exception is an event which disturbs the flow of execution
- During method execution, if an event disturbs the execution the method creates an object of Throwable type and it is handed over to the run time system. This phenomenon is known as Raise of Exception.
- The run Time system looks for a block of code which is capable of handling the events
- The block of code which handles the events is known as exception handler.
- The Run Time System looks for the handler in the current method context, if the handler is not specified it looks for the handler in the caller method, likewise the run time system looks for the handler in the entire **call Stack**, if it can't find the handler the run time system terminates the entire system abruptly which might results in losing the data.

Exception Hierarchy:-



CORE JAVA- PART III

■ **Program to illustrate the Exception Handling**

```
package pack1;

public class Demo1 {


	public static void main(String[] args) {

		int x=12;
		int y=0;
		try {
			System.out.println("Try Block Started");
			y=x/0;
			System.out.println("Try Block Ended");

		} catch (ArithmeticException e) {

			System.out.println("cann't divide by Zero");
		}
		System.out.println("x = "+x);
		System.out.println("y = "+y);
	}

}
```

Output:

```
Try Block Started
cann't divide by Zero
x = 12
y = 0
```

■ Program to illustrate interrupted Exception

```
package pack1;

public class Demo3 {

	public static void main(String[] args) {

		try {
		for (int i = 0; i < 25; i++) {
			System.out.println("i= "+i);
			Thread.sleep(1000);
		}
		}
		catch (InterruptedException e) {
			System.out.println("sleep is interrupted...");
		}
	}

}
```

■ Output

```
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10
i= 11
i= 12
i= 13
i= 14
i= 15
i= 16
i= 17
i= 18
i= 19
i= 20
i= 21
i= 22
i= 23
i= 24
```

- The Exceptions which are checked and anticipated by the compiler is known as checked exception
- During compilation of the source code if compiler checks for an exception, it is compulsory to define the Handler. Otherwise the compiler throws error.
- All exception classes which are direct sub class to the exception class except run time exception are known as checked exception.
- The exception which are not anticipated by the compiler is known as unchecked exception
- The unchecked will not be known until we run the program or until the operation happens.
- All the exception which are sub class to run time exception class are kwon as unchecked exception.
- The checked exception are known at compile time occur at run time.
- Whereas unchecked exception are kwon at run time and occurs at run time.

- Java program to illustrate ClassFoundException

```
package pack1;

public class Demo4 {

	public static void main(String[] args) {


		try {
			Class.forName("demo1");
		} catch (ClassNotFoundException e) {
```

```
                System.out.println("class not found in the project");
                }
        }

}
```

- Java Program to illustrate the multiple catch block

```
package pack1;

class Demo1 {
        public static void main(String[] args) {
                int x=12;
                int y=0;
                int a[]=new int[5];
                try {
                        System.out.println("Try Block Started");
                        y=x/2;
                        a[6]=56;
                        System.out.println("Try Block Ended");
                } catch (ArithmeticException e) {
                        System.out.println("can't divide by Zero");
                }
                catch (ArrayIndexOutOfBoundsException e) {
                        System.out.println("array index is out of Bounds");
                }
                System.out.println("x = "+x);
                System.out.println("y = "+y);
        }
}
```

Output:

```
Try Block Strated
array index is out of Bounds
x = 12
y = 6
```

- Java program to illustrate Nested try-catch Block

```
package pack1;

public class Demo5 {
        public static void main(String[] args) {
                        int x=12;
                        int y=0;
                        int a[]=new int[5];
                        try {
                                System.out.println("Try Block Started");

                                try{
                                        y=x/0;
                                }catch (ArithmeticException e) {
                                        System.out.println("can't divide by Zero");

                                a[6]=56;
```

```
                        System.out.println("Try Block Ended");
                    }
                    }

                    catch (ArrayIndexOutOfBoundsException e) {
                        System.out.println("array index is out of Bounds");
                    }
                    System.out.println("x = "+x);
                    System.out.println("y = "+y);
                }


}
```

OUTPUT:
Try Block Started
can't divide by Zero
array index is out of Bounds
x = 12
y = 0

Note:

- ➔ final – is a keyword
- ➔ finally- is a block
- ➔ finalize()- is a function

■ Program to illustrate the try-catch-finally block

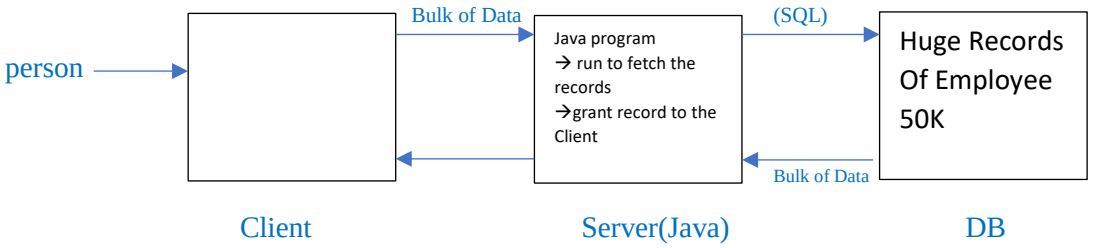
```
package pack1;

public class Demo6 {
    public static void main(String[] args) {
        int x=12;
        int y=0;
        int a[]=new int[5];
        try {
            System.out.println("Try Block Started");

            try{
                y=x/0;
            }catch (ArithmeticException e) {
                System.out.println("can't divide by Zero");

            a[6]=56;
            System.out.println("Try Block Ended");
        }
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("array index is out of Bounds");
        System.exit(0);
        }
        finally
        {
            System.out.println("I Run Always");
        }
        System.out.println("x = "+x);
        System.out.println("y = "+y);
    }
```

}

OUTPUT:
Try Block Started
can't divide by Zero
array index is out of Bounds
I Run Always
x = 12
y = 0

Backup class Notes:

For Ex. Increase the 5% sal of the Emp Record



Collection framework Library :

Queue → Ex. Music Player

Stack → Ex. Gmail Inbox

Tree→ Ex. File System in Computer

Collections:

- Representing a group of Object as a Single entity is known as collection.
- The object might be of same type or different type.
- The collection doesn't have any limitation in size
- It grows dynamically.
- Ex. A group of Songs in a Playlist is a collection.
- A group of mails in inbox is a collection
- A group of Apps in Android iOS Phone is a collection.

Collection Framework Library:-

- CFL is a Set of Interfaces and classes which implements the Standard data structures like
  i. List
  ii. Queue
  iii. Set
  iv. Map
  v. And other
- The library provides set of implementation classes where programmer can make use of it to build required DS
- The Library also provides set of implemented algorithms for accessing the elements, for sorting, and for searching.

CORE JAVA- PART III

- The benefit of this library is
  - ✓ Accurate
  - ✓ Robust
  - ✓ Easy to use
  - ✓ Customizable
  - ✓ Portable

1. Add operation
   - ✓ Adding an element (object) into the collection
   - ✓ Element is casted to Object class type (Upcasting)
   - ✓ Elements present inside the collection shows Object class Properties
2. Remove operation
   - ✓ Getting access to the element present inside the collection
   - ✓ Down casting to get Original properties
3. Remove operation
   - ✓ Removing the elements from the collection
   - ✓ Remove object shows object class property
   - ✓ Next down casted get original properties

Types of collections:

1. List
2. Queue
3. Set

- List stores the element with index
- Queue stores elements without index but its processed in FIFO
- Set stores elements without index, No duplicates in the set

---

1. Specific Exception

```
try {
        } catch (ArithmeticException e) {
                e.printStackTrace();
        }
```

2. RuntimeException

```
        try {
        } catch (RuntimeException e) {
                e.printStackTrace();
        }
```

CORE JAVA- PART III

3. All type of Exception Handler:

```
try {

        } catch (Exception e) {

                e.printStackTrace();

        }
```

4. Throwable type Handler

```
try {

        } catch (Throwable e) {

                e.printStackTrace();

        }
```

5. Catch block for Exception should be always written at last

```
try {

                System.out.println(2/0);

        } catch (ArithmeticException e) {

                e.printStackTrace();
        }
        catch (NumberFormatException e) {
                e.printStackTrace();
        }
        catch (Exception e) {
                e.printStackTrace();
                }
```

6. Unreachable catch block for FileAlreadyExistsException. It is already handled by the catch block for Exception

```
try {

                System.out.println(2/0);

        } catch (ArithmeticException e) {

                e.printStackTrace();
        }
        catch (NumberFormatException e) {
                e.printStackTrace();
        }
        catch (Exception e) {
                e.printStackTrace();
        }
        catch (FileAlreadyExistsException e) {

                        }
```

---

1. Throw Statement

CORE JAVA- PART III

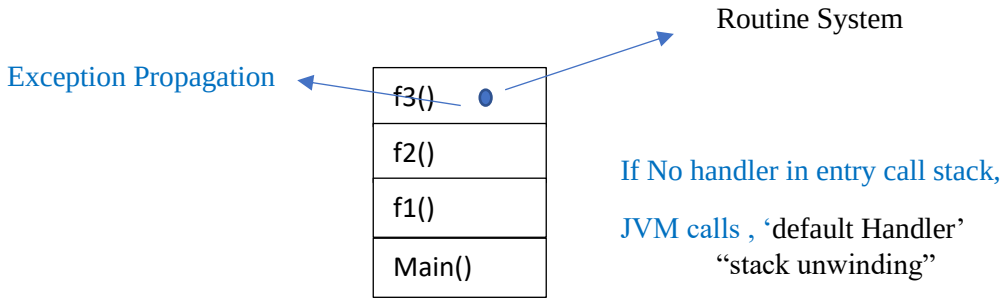
- Used to throw an Object
  Syntax: throw e;
  e--> must be an object of Exception type
- Throw -->can throw only one Object at a time
- Throw statement must be used inside
  i. method body
  ii. constructor Body
    1. Example

```
throw new ArithmeticException();
```

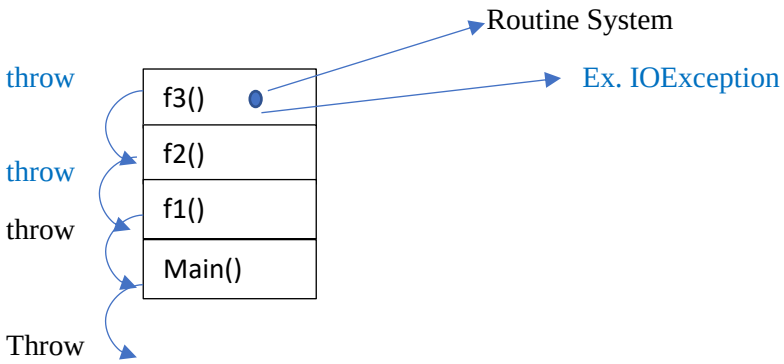
    2. **Example**

```
ArithmeticException e1=new ArithmeticException();
throw e1;
```

➔ Method throwing unchecked Exception



Method throwing checked Exception:



■ Program to illustrate the Exception Propagation

```
package pack1;

public class Calculator {


    void divide(int n1,int n2)
    {

        System.out.println("Dividing by "+n1+" by "+n2);
        int res=n1/n2;
        System.out.println("Result is "+res);
    }
```

```
}
```

```
package pack1;

import java.util.Scanner;

public class Demo8 {

	public static void main(String[] args) {

		Scanner scan=new Scanner(System.in);

		System.out.println("Enter numerator value ");
		int numera=scan.nextInt();
		System.out.println("Enter denominator value");
		int deno=scan.nextInt();

		Calculator c1=new Calculator();
		try {
			c1.divide(numera, deno);


		} catch (ArithmeticException e) {
			System.out.println("unable to divide by 0");
			e.printStackTrace();
		}

	}

}
```

OUTPUT:

```
Enter numerator value
12
Enter denominator value
0
Dividing by 12 by 0
java.lang.ArithmeticException: / by zero
unable to divide by 0
	at pack1.Calculator.divide(Calculator.java:10)
	at pack1.Demo8.main(Demo8.java:18)
```

- **program to illustrate the throws keyword**

```
package pack1;

public class Sequence {


	void printNumbers(int start,int end) throws InterruptedException

	{
		for (int i = start; i <= end; i++) {
			System.out.println(i);
			Thread.sleep(1000);
```

```
		}
	}
}
```

```
package pack1;

public class Demo10 {

	public static void main(String[] args) {

		Sequence sq=new Sequence();



		try {
			sq.printNumbers(100, 110);
		} catch (InterruptedException e) {

			e.printStackTrace();
		}
	}
}
```

Output:

```
100
101
102
103
104
105
106
107
108
109
110
```

- If a method is throwing unchecked exception , the runtime system looks for the handler in the current method itself.
- If the handler is not specified the unchecked propagates implicitly to the caller function looking for the Handler. Like wise the unchecked exception looks for the handler in the entire call trace, if no handler is found the JVM calls the default handler which terminates the execution, before terminating prints the details about the exception.
- The forcible collapsing of stack because of not handling the Stack is known as Stack unwinding
- If the method throws the checked exception type object it must be handled in the method body where the exception occurs. Because it cant propagate the caller function
- The checked Exception must be handled then and there itself or it can be thrown to the caller by using throws declaration keyword.

Throws Keyword:

- Throws keyword is a declaration keyword which must be used in either in the method declaration or in the constructor declaration
- It declares the list of exceptions which is not handled in the current method body

CORE JAVA- PART III

- The throws keyword specifies the list of exception.
- It is a specification keyword which will be specifying the caller that mention the exception are not handled in the current method context so that caller method can define the handler
- It is left to the caller method whether to handler the exception by defining try-catch block or specifies the exception list to its caller
- The throws keyword must be used only for the checked exception category.

- Program to illustrate another way of handling multiple Exception

---

```
package pack1;

import java.sql.SQLException;

public class Demo10	{

public static void main(String[] args) {

		Sequence sq=new Sequence();



		try {
			sq.printNumbers(100, 110);
		} catch (InterruptedException |SQLException e) {

			e.printStackTrace();
		}
		}
}
```

- Program illustrate the Exception in overriding a method

```
package pack1;

import java.io.FileNotFoundException;

public class Sample1 {


	void test() throws Exception
	{
		//write code to do operation
	}



}
class Sample2
{

	void test() throws FileNotFoundException
	{
		//override the test method implementation
```

```
    }

        }
```

- When overriding a method, in the subclass the overridden method should not declare or specify any exception if in super class is not declaring or specifying any exception
- If the super class method declares or specifies an exception while overriding the sub class should declare or specify same exception or its sub class exception
- If we violate these rules the compiler throws error.

Program to illustrate the Exception in interface

```
package pack1;

public interface SequenceNumber {

    void generateSeries (int start,int end) throws InterruptedException;

}
class SequenceNumberImpl implements SequenceNumber
{

    @Override
    public void generateSeries(int start, int end) throws InterruptedException {

        for (int i = start; i <= end; i++) {
            System.out.println(i);
            Thread.sleep(1000);

        }

    }

}
```

Program to illustrate the return values in try-catch block

```
package pack1;

public class Sample3 {

    int doOperation()
    {
        try {

            return 1;

        } catch (Exception e) {


            return 2;
        }

        return 3; // ERROR-Unreachable code
```

```
	}

}
```

- Program to illustrate Exceptions in try-catch-finally Block

```
package pack1;

public class Sample3 {

	int doOperation()
	{
		try {


			int res=12/0;
			System.out.println("try block");
			return 1;

		} catch (Exception e) {


			System.out.println("catch block");
		return 2;
		}
		finally {
			System.out.println("I run Always-finally");


		return 3;

		}

	}

}
package pack1;

public class MainClass1 {

	public static void main(String[] args) {
		Sample3 s=new Sample3();

		int retValue=s.doOperation();
		System.out.println(retValue);
	}

}
```

OUTPUT:
catch block
I run Always-finally
3

- How to throw an Exception?

CORE JAVA- PART III

```
package pack1;

public class Calculator2 {
void divide(int n1,int n2)
{
	if(n2==0)
	{
		throw new ArithmeticException("denominator can not be zero");
	}else
	{
		int Res=n1/n2;
		System.out.println("result = "+Res);
	}
}
}
```

```
package pack1;

public class MainClass3 {
	public static void main(String[] args) {

		Calculator2 calc=new Calculator2();
		try {
			calc.divide(15, 0);
		} catch (ArithmeticException e) {
			String msg=e.getMessage();
			System.out.println(msg);
		}

	}

}
```

Output:
denominator can not be zero

- ?

```
package pack1;

import java.sql.SQLException;

public class MainClass3 {
	public static void main(String[] args) throws SQLException {

		Sample4 s=new Sample4();
		s.test();

	}

}
```

- Program to illustrate User-defined Exception

CORE JAVA- PART III

```
package pack1;

public class InsufficientBalanceException extends Exception {

	private String msg;

	public InsufficientBalanceException()
	{

	}

	public InsufficientBalanceException(String msg) {

		this.msg = msg;
	}

	public String getMessage() {
		return this.msg;
	}



}

package pack1;

public class Account {
	double accbal;
	String name;
	public Account(double accbal, String name) {

		this.accbal = accbal;
		this.name = name;
	}
	void deposit(double amt)
	{
		System.out.println("Depositing Rs."+amt);
		accbal=accbal+amt;

	}
	void withdraw(double amt) throws InsufficientBalanceException
	{
		if(amt>accbal) {
			throw new InsufficientBalanceException("account balance is not sufficient");
		}
		System.out.println("withdrawing RS. "+amt);
		accbal=accbal-amt;
	}
	void balanceEnquery() {
		System.out.println("Acoount balance is Rs. "+accbal);
	}

}

package pack1;

public class MainClass5 {

	public static void main(String[] args) {
		Account a1=new Account(5000.00, "hdb");
```

```
                a1.balanceEnquery();
                a1.deposit(1000.00);
                a1.balanceEnquery();
                try {
                        a1.withdraw(7000.00);
                } catch (InsufficientBalanceException e) {
                System.out.println(e.getMessage());
                }
                a1.balanceEnquery();
        }
}
```

OUTPUT:
Acoount balance is Rs. 5000.0
Depositing Rs.1000.0
Acoount balance is Rs. 6000.0
account balance is not sufficient
Acoount balance is Rs. 6000.0

1. Single Task Application
   Ex. Calculator, MS-DOS → command prompt
2. Multi-Tasking Applications
   Ex. Windows, MS-word, music player ext.

1. Process- heavy weight, consumes more memory, slow
2. Threads- light weight, less memory, faster

- In java, the execution of program takes in the JVM, whenever we provide a class for execution to the JVM , the JVM creates a thread and runs the program
- JVM uses threads to executes in order to achieve concurrency or running more than one program in parallel.
- In Java, if we have to achieve to concurrency then we should run program in threads.
- A thread is a instance or object of Thread class.

Thread Class:

- Thread is a class defined in java.lang package which defines the thread properties and thread behaviors so that the JVM can run the thread in a concurrent mode.
- The thread properties are
  i. Thread id
     - This is used to identify each thread uniquely.
     - It is a integer type property, which is initialized by default.
     - We can not change the id of the thread, we can retrieve the id but we cant set the id value

  ii. Thread name:
     - This property is used to provide a name to the thread created in the JVM
     - The name property can be provided by the programmer and it can be changed at any time
     - If we don't initialize it, the default initialization is provided

  iii. Thread priority:
     - It is a integer number which is used to set the propriety of a thread
     - The programmer can define the thread priority for his thread. The thread priority must be within the range 1 to 10.
     - 1 is lowest and 10 is highest
     - If we don't provide the default priority is 5.
- These properties are private member variables of the thread class, the thread class provides getters and setters

Methods of thread class:

CORE JAVA- PART III

1. Public in getId()
   On invoking this method, it returns the id of the thread
2. Public int getPriority()
   - On invoking this method, it returns the priority of the thread
3. Public String getNAme()
   - On invoking this method it retuens the name of the thread
4. Public void setPriority(int priorityNo)
   - The method sets the priority of the thread to the specified priority number
5. Public void setName(String threadName)
   - The method sets the name of the thread to the specified thread name
6. Public void run()
   - The run method is used to define the Task of a thread
   - In the thread class, this method has a empty implementation.
7. Public void start()
   - On invoking this method it begins the execution of thread. It allocates a new stack for execution and internally calls run the run method of the object to perform the task.
8. Public void stop()
   - On invoking this method , it stops the execution of the thread. It is not compulsory to invoke this method in order to stop the execution, if running completes the task it automatically stops the execution.
9. Public static Thread currentThread()
   - On invoking this method, it returns the reference to the current running thread.
10. Public void static sleep(long time)
    - On invoking this method, it causes the thread to pass the execution for the specified time.
    - The current thread goes to sleep mode.

Thread Class Constructor:

1. No arg constructor

Thread()

➢ If thread arg is created the no arg constructor , the thread properties will be interlaced to the default values.

2. String type arg constructor

Thread(String arg)

➢ Creates a thread class object with the thread name provided in the constructor.

3. Runnable type arg constructor

Thread(Runnable arg)

➢ Creates a thread Object by using runnable type Object.

4. Two arg constructor

First arg- Runnable type
Second arg – String type
Thread(Runnable arg, String arg)

➢ Creates a thread Object with specified Runnable type Object and with the specified thread name.

- Program to illustrate threads

CORE JAVA- PART III

```
package pack1;

public class Demo1 {

	public static void main(String[] args) {


		Thread t1=new Thread();
		System.out.println("thread ID "+t1.getId());
		System.out.println("Thread name "+t1.getName());
		System.out.println("thread Prority "+t1.getPriority());
	System.out.println("modifyimg thread properties");
	t1.setName("MyThread");
	t1.setPriority(10);

	System.out.println("thread ID "+t1.getId());
	System.out.println("Thread name "+t1.getName());
	System.out.println("thread Prority "+t1.getPriority());



	}


}
```

OUTPUT:

```
thread ID 10
Thread name Thread-0
thread Prority 5
modifyimg thread properties
thread ID 10
Thread name MyThread
thread Prority 10
```

- Program to illustrate Multiple threads creation

```
package pack1;

public class SequenceThread extends Thread{


	public void run()
	{
		System.out.println("printig number from 1 to 25");
		for (int i = 1; i < 25; i++) {
			System.out.println("i ="+i);
			try {
				Thread.sleep(1000);

			} catch (InterruptedException e) {

				e.printStackTrace();
```

```
                        }
                }
        }

}
package pack1;

public class MainClass {


        public static void main(String[] args) {

                System.out.println("main method started");
                SequenceThread s=new SequenceThread();
                s.start();
                SequenceThread s1=new SequenceThread();
                s1.start();
                System.out.println("main method ended");
        }

}
main method started
main method ended
printig number from 1 to 25
printig number from 1 to 25
i =1
i =1
i =2
i =2
i =3
i =3
i =4
i =4
i =5
i =5
i =6
i =6
i =7
i =7
i =8
i =8
i =9
i =9
i =10
i =10
i =11
i =11
i =12
i =12
i =13
i =13
i =14
i =14
i =15
i =15
i =16
i =16
i =17
i =17
```

```
i =18
i =18
i =19
i =19
i =20
i =20
i =21
i =21
i =22
i =22
i =23
i =23
i =24
i =24
```

```
package pack1;

public class SequenceThread2 extends Thread {




	public void run()
	{
		System.out.println("printig number from 101 to 125");
		for (int j = 101; j < 125; j++) {
			System.out.println("j ="+j);
			try {
				Thread.sleep(1000);

			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		}
	}

}
package pack1;

public class MainClass {


	public static void main(String[] args) {

		System.out.println("main method started");
		SequenceThread s=new SequenceThread();
		s.start();
		SequenceThread2 s1=new SequenceThread2();
		s1.start();
		System.out.println("main method ended");
	}

}
```

```
OUTPUT:
main method started
printig number from 1 to 25
```

```
i =1
main method ended
printig number from 101 to 125
j =101
j =102
i =2
i =3
j =103
j =104
i =4
j =105
i =5
j =106
i =6
i =7
j =107
j =108
i =8
i =9
j =109
j =110
i =10
j =111
i =11
j =112
i =12
j =113
i =13
j =114
i =14
i =15
j =115
j =116
i =16
j =117
i =17
j =118
i =18
j =119
i =19
i =20
j =120
j =121
i =21
j =122
i =22
i =23
j =123
j =124
i =24
```

- Program to illustrate invoking of riun() method

```
package pack1;

public class MainClass {
```

```
	public static void main(String[] args) {

		System.out.println("main method started");
		SequenceThread s=new SequenceThread();
		s.run();
		SequenceThread2 s1=new SequenceThread2();
		s1.run();
		System.out.println("main method ended");
	}

}
main method started
printig number from 1 to 25
i =1
i =2
i =3
i =4
i =5
i =6
i =7
i =8
i =9
i =10
i =11
i =12
i =13
i =14
i =15
i =16
i =17
i =18
i =19
i =20
i =21
i =22
i =23
i =24
printig number from 101 to 125
j =101
j =102
j =103
j =104
j =105
j =106
j =107
j =108
j =109
j =110
j =111
j =112
j =113
j =114
j =115
j =116
j =117
j =118
j =119
j =120
```

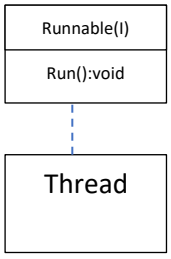
```
j =121
j =122
j =123
j =124
main method ended
```

- Instead of invoking start() method, we can invoke run method on a thread, it performs the execution on the current stack hence No concurrency.
- We get concurrency execution mode only when we call start() method on a thread.

1. Can we override start() method?
   Yes, we can override the start method because it's a non-static method but if we override we lose the implementation of stack allocation and calling the main method.

Runnable Interface:

- Defined in java.lang package
- It specifies or defined the main method
  Public void run()
- A class which implements runnable interface is eligible to run on stack
- Thread class is a implementation class of runnable interface.



- Program to illustrate Runnable Interface

J

```
package pack1;

public class Sequence3 implements Runnable{


	public void run() {
		System.out.println("printig number from 1 to 25");
		for (int i = 101; i < 125; i++) {
			System.out.println("i ="+i);
			try {
				Thread.sleep(1000);

			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		}

	}
}
```

OUTPUT:

CORE JAVA- PART III

```
main method is started
main method is ended
printig number from 1 to 25
i =101
i =102
i =103
i =104
i =105
i =106
i =107
i =108
i =109
i =110
i =111
i =112
i =113
i =114
i =115
i =116
i =117
i =118
i =119
i =120
i =121
i =122
i =123
i =124
```

CORE JAVA- PART III

- Program to illustrate Thread Safe

```
package pack1;

public class SharedResource {

	synchronized void resource1()
	{
		String thName=Thread.currentThread().getName().toUpperCase();
		System.out.println(thName+"printig number from 1 to 25");
		for (int i = 1; i < 25; i++) {
			System.out.println(thName+"i ="+i);
			try {
				Thread.sleep(1000);

			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		}
	}

	synchronized void resource2()
	{
		String thName=Thread.currentThread().getName().toUpperCase();
		System.out.println(thName+"printig number from 101 to 125");
		for (int i = 101; i < 125; i++) {
			System.out.println("i ="+i);
			try {
				Thread.sleep(1000);

			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		}
	}

}
package pack1;

public class ThreadA extends Thread{


	SharedResource sr;

	public ThreadA(SharedResource sr) {

		this.sr = sr;
	}

	public void run()
	{
	sr.resource1();
	}
```

```
}
package pack1;

public class StartThreads {


	public static void main(String[] args) {

		System.out.println("HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH");
		SharedResource sr1=new SharedResource();

		ThreadA t1=new ThreadA(sr1);
		ThreadA t2=new ThreadA(sr1);
		ThreadA t3=new ThreadA(sr1);

		t1.setName("T1-thread");
		t1.setName("T2-thread");
		t1.setName("T3-thread");

		t1.start();
		t2.start();
		t3.start();









		System.out.println("HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH");

	}

}
```

OUTPUT:

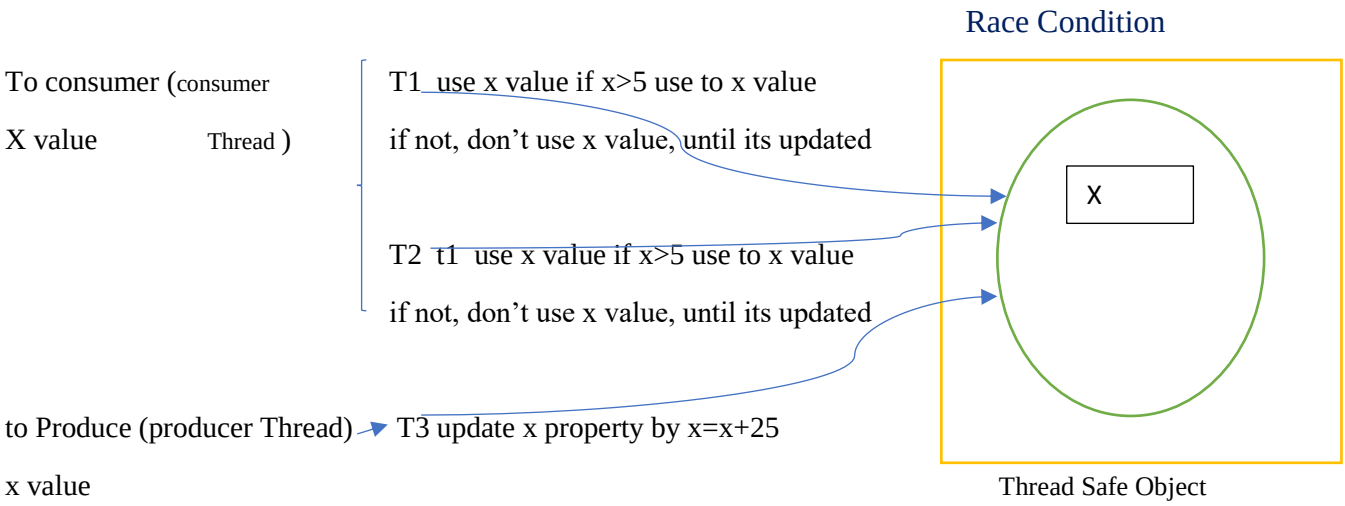
```
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
T3-THREAD printig number from 1 to 25
T3-THREAD i =1
T3-THREAD i =2
T3-THREAD i =3
T3-THREAD i =4
T3-THREAD i =5
T3-THREAD i =6
T3-THREAD i =7
T3-THREAD i =8
T3-THREAD i =9
T3-THREAD i =10
T3-THREAD i =11
T3-THREAD i =12
T3-THREAD i =13
T3-THREAD i =14
T3-THREAD i =15
T3-THREAD i =16
T3-THREAD i =17
T3-THREAD i =18
T3-THREAD i =19
```

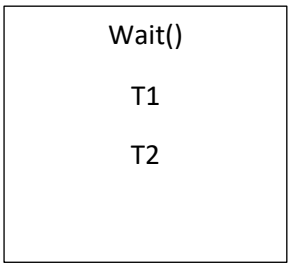
```
T3-THREAD i =20
T3-THREAD i =21
T3-THREAD i =22
T3-THREAD i =23
T3-THREAD i =24
THREAD-2 printig number from 1 to 25
THREAD-2 i =1
THREAD-2 i =2
THREAD-2 i =3
THREAD-2 i =4
THREAD-2 i =5
THREAD-2 i =6
THREAD-2 i =7
THREAD-2 i =8
THREAD-2 i =9
THREAD-2 i =10
THREAD-2 i =11
THREAD-2 i =12
THREAD-2 i =13
THREAD-2 i =14
THREAD-2 i =15
THREAD-2 i =16
THREAD-2 i =17
THREAD-2 i =18
THREAD-2 i =19
THREAD-2 i =20
THREAD-2 i =21
THREAD-2 i =22
THREAD-2 i =23
THREAD-2 i =24
THREAD-1 printig number from 1 to 25
THREAD-1 i =1
THREAD-1 i =2
THREAD-1 i =3
THREAD-1 i =4
THREAD-1 i =5
THREAD-1 i =6
THREAD-1 i =7
THREAD-1 i =8
THREAD-1 i =9
THREAD-1 i =10
THREAD-1 i =11
THREAD-1 i =12
THREAD-1 i =13
THREAD-1 i =14
THREAD-1 i =15
THREAD-1 i =16
THREAD-1 i =17
THREAD-1 i =18
THREAD-1 i =19
THREAD-1 i =20
THREAD-1 i =21
THREAD-1 i =22
THREAD-1 i =23
THREAD-1 i =24
```

CORE JAVA- PART III

- If a thread enters synchronized non static method creates an object lock and utilizes the Object property.
- After the execution the thread releases the lock so that other thread can use the object properties.
- When an object is locked by a thread the other threads will be in the blocked state until the current thread releases the lock
- If current thread doesn't releases the lock there it leads thread Dead Lock
- If a thread a enters synchronized static method it creates class lock.

ITC- inter thread communication





notify() and notifyAll()

```
package pack2;

public class Data
{
	private int x=4;

	public int getX() {
		return x;
	}
	synchronized public void consumeX(int arg)
	{
		String th_name= Thread.currentThread().getName();
		System.out.println(th_name+"  running ");
		if(x < 5){
```

```
                System.out.println("insufficient data , waiting for
notification");
                 try {
                      wait();
                } catch (InterruptedException e) {
                      e.printStackTrace();
                }
                 System.out.println(th_name+" resuming back");
                 x=x-arg;
                 System.out.println("x valu by "+th_name+"="+x);
          }else{
                System.out.println("data sufficient");
                x=x-arg;
          }
     }
     synchronized public void produceX(int arg)
     {
          String th_name= Thread.currentThread().getName();
          System.out.println(th_name+" producing x values");
          x=x+arg;
          System.out.println(th_name+" notifying the threads...");
          notifyAll();
     }

}
package pack2;

public class ProduceThread extends Thread {
     private Data d1;

     public ProduceThread(Data d1) {
          this.d1=d1;
     }

     public void run()
     {
          d1.produceX(20);
     }

}


package pack2;

public class ConsumerThread extends Thread
{

     private Data d1;
```

```
	public ConsumerThread(Data d1) {
		this.d1=d1;
	}

	public void run()
	{
		d1.consumeX(8);
	}

}
package pack2;

public class StartThreads
{
	public static void main(String[] args) {
		System.out.println("main method started");
		Data datObj = new Data();
		ConsumerThread t1 = new ConsumerThread(datObj);
		ConsumerThread t2 = new ConsumerThread(datObj);
		ConsumerThread t3 = new ConsumerThread(datObj);
		ProduceThread t4 = new ProduceThread(datObj);

		t1.setName("T1-Thread");
		t2.setName("T2-Thread");
		t3.setName("T3-Thread");
		t4.setName("T4-Thread");


		t1.start();
		t2.start();
		t3.start();

		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		t4.start();
		System.out.println("x value is : "+datObj.getX());
		System.out.println("main method ended");
	}
}
OUTPUT:
main method started
T1-Thread  running
insufficient data , waiting for notification
T2-Thread  running
insufficient data , waiting for notification
T3-Thread  running
```

```
insufficient data , waiting for notification
x value is : 4
T4-Thread producing x values
main method ended
T4-Thread notifying the threads...
T3-Thread resuming back
x valu by T3-Thread=16
T2-Thread resuming back
x valu by T2-Thread=8
T1-Thread resuming back
x valu by T1-Thread=0
```

Object Cloning:

Public Object clone()

- Method is declared in object class
- It is implemented to take a clone of the object on which method is invoked
- The method is a shallow copy cloning
- We can take class of a object if it is a type of cloneable.

- Program to illustrate Object Cloning

```
package pack1;

public class Mobile implements Cloneable {

	int imei;
	String Os;
	int RamSize;
	public Mobile(int imei, String os, int ramSize) {
		super();
		this.imei = imei;
		Os = os;
		RamSize = ramSize;
	}
	@Override
	public String toString() {
		return "Mobile [imei=" + imei + ", Os=" + Os + ", RamSize=" + RamSize + "]";
	}
	@Override
	protected Object clone() throws CloneNotSupportedException {
```

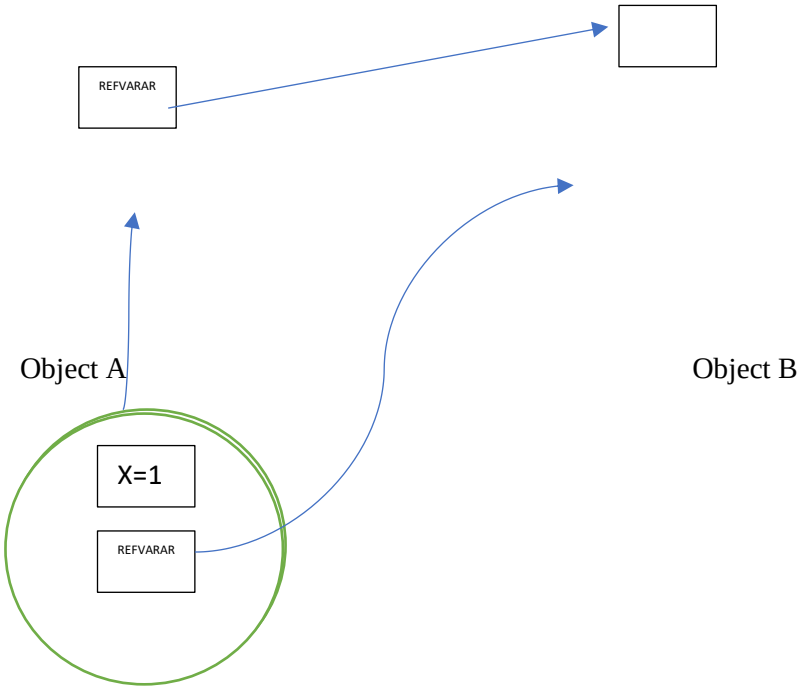
CORE JAVA- PART III

```
		return super.clone();
	}

}
package pack1;



public class MainClass1 {

	public static void main(String[] args) {

		Mobile OrginalMobile=new Mobile(1234, "android", 4);
		Mobile cloneMobile=null;
		System.out.println(OrginalMobile);

		try {
			cloneMobile=(Mobile) OrginalMobile.clone();
			System.out.println("cloned");
		} catch (CloneNotSupportedException e) {
			System.out.println("can not take a clone of th
object");
		}

		System.out.println(cloneMobile);
		System.out.println(OrginalMobile.hashCode());
		System.out.println(cloneMobile.hashCode());


	}
}
```

```
OUTPUT:
Mobile [imei=1234, Os=android, RamSize=4]
cloned
Mobile [imei=1234, Os=android, RamSize=4]
366712642
1829164700
```

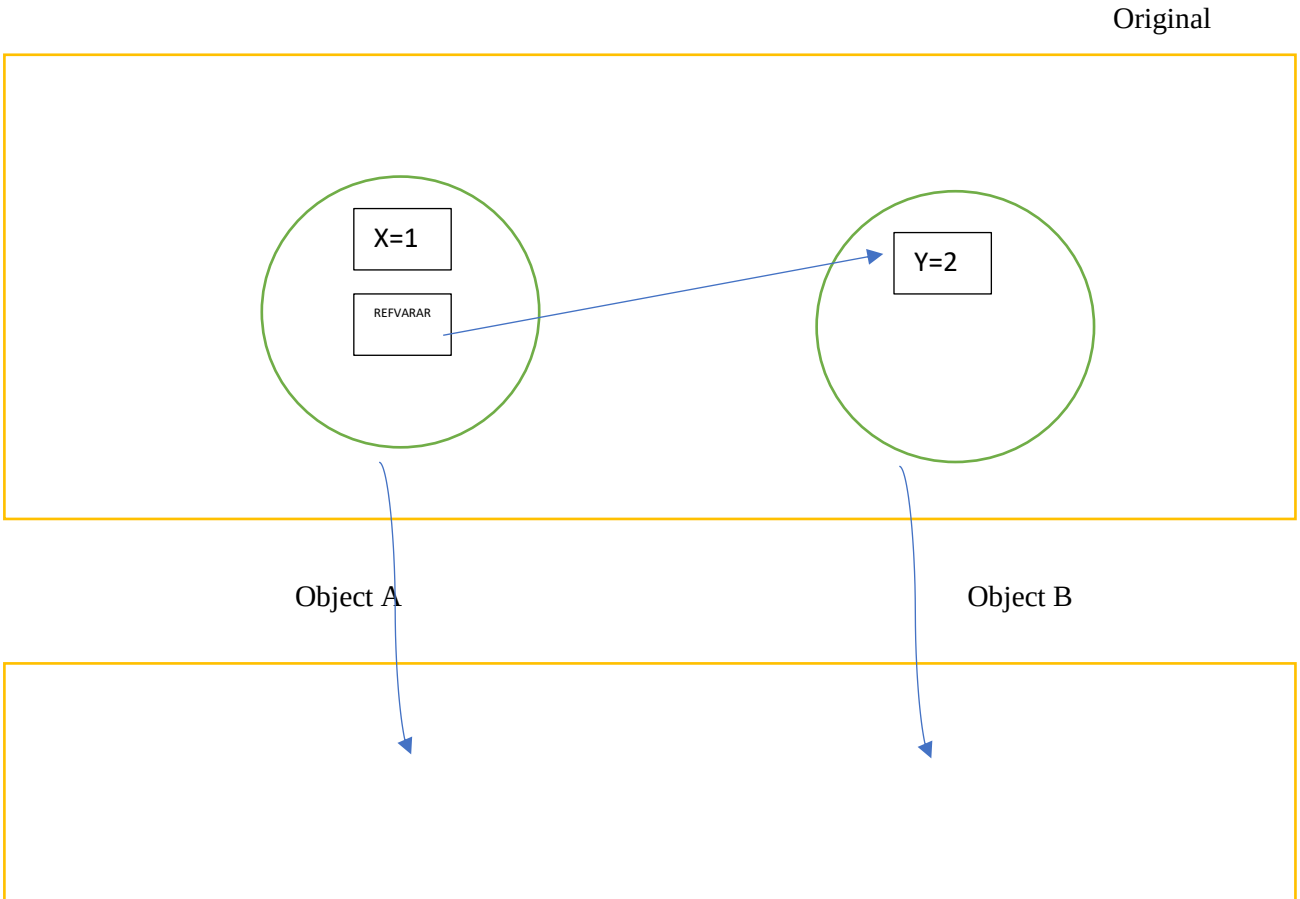
Shallow Copy:

CORE JAVA- PART III

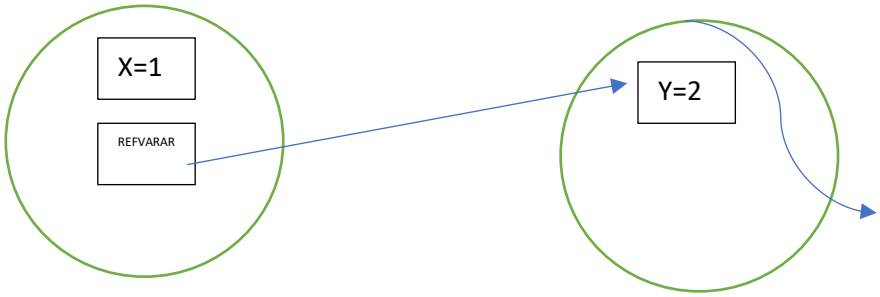


Deep Copy:

Shallow Copy:



CORE JAVA- PART III



- Program to show Deep Cloning

```
package pack1;

public class Mobile implements Cloneable {

	int imei;
	String Os;
	int RamSize;
	Battery b1=new Battery(123456,1000);
	public Mobile(int imei, String os, int ramSize) {
		super();
		this.imei = imei;
		Os = os;
		RamSize = ramSize;
	}
	@Override
	public String toString() {
		return "Mobile [imei=" + imei + ", Os=" + Os + ", RamSize=" +
RamSize + "]";
	}
	@Override
	protected Object clone() throws CloneNotSupportedException {
Mobile m1=new Mobile(this.imei, this.Os, this.RamSize);
Battery b2=new Battery(this.b1.number, this.b1.capacity);
m1.b1=b2;
return m1;


	}

}
package pack1;
```

CORE JAVA- PART III

```
public class Battery {
	int number;
	int capacity;
	public Battery(int number, int capacity) {

		this.number = number;
		this.capacity = capacity;
	}
	@Override
	public String toString() {
		return "Battery [number=" + number + ", capacity=" + capacity + "]";
	}




}
package pack1;



public class MainClass1 {

	public static void main(String[] args) {

		Mobile OrginalMobile=new Mobile(1234, "android", 4);
		Mobile cloneMobile=null;
		System.out.println(OrginalMobile);

		try {
			cloneMobile=(Mobile) OrginalMobile.clone();
			System.out.println("cloned");
		} catch (CloneNotSupportedException e) {
			System.out.println("can not take a clone of th object");
		}

		System.out.println(cloneMobile);
		System.out.println("hashCode number for orginal number =
"+OrginalMobile.hashCode());
		System.out.println("hashCode number for clone mobile =
"+cloneMobile.hashCode());


		System.out.println("hashCode for Battery =
"+OrginalMobile.b1.hashCode());
		System.out.println("hashCode for battery =
"+cloneMobile.b1.hashCode());
		System.out.println(OrginalMobile.b1);
		System.out.println(cloneMobile.b1);

		System.out.println(OrginalMobile);

	}
}
OUTPUT:
Mobile [imei=1234, Os=android, RamSize=4]
cloned
Mobile [imei=1234, Os=android, RamSize=4]
```

CORE JAVA- PART III

```
hashCode number for orginal number = 366712642
hashCode number for clone mobile = 1829164700
hashCode for Battery = 2018699554
hashCode for battery = 1311053135
Battery [number=123456, capacity=1000]
Battery [number=123456, capacity=1000]
Mobile [imei=1234, Os=android, RamSize=4]
```

Type Unsafety:

Variables:

Int x;

Double y;

String s1;

Type variablename;

1. Class type
2. Interface type
3. Enum type

Eg. Age= 10
Age= 10
Age= 100
Age= 1000 (not a valid data)
Ex. Month= (1~12)

```
package pack1;

public enum Day {

	MONDAY,TUESDAY,WEDNESDY,THURSDAY,FRIDAY,SATURDAY,SUNDAY;

}
```

```
package pack1;

public class mainclass {
	public static void main(String[] args) {
		Day dayname;
		dayname=Day.THURSDAY;
		System.out.println("Today is "+dayname);
	}

}
```

OUTPUT:

CORE JAVA- PART III

```
Today is THURSDAY
```

Interface type → specify the behavior (what)

Class type → define the behavior (how)

Enum type→ define the set of constant

When and where → Polymorphism

```
package pack1;

public class Student<T> {
T id;

public Student(T id) {

	this.id = id;
}

public T getId() {
	return id;
}

public void setId(T id) {
	this.id = id;
}

}
```

```
package pack1;

public class Student<T> {
T id;

public Student(T id) {

	this.id = id;
}

public T getId() {
	return id;
}

public void setId(T id) {
	this.id = id;
}

}
```

```
package pack1;

public class MainClass3 {
	public static void main(String[] args) {
```

```
		Student<Integer> s1=new Student<Integer>(1234);

		System.out.println("Student id : "+s1.getId());

		Student<String> s2=new Student<String>("4ub13ec018");

		System.out.println("Student id : "+s2.getId());




	}

}
```

OUTPUT:

```
Student id : 1234
Student id : 4ub13ec018
```

Generics:

```
package pack1;

import java.util.ArrayList;

public class MainClass4 {

	public static void main(String[] args) {

		ArrayList l1=new ArrayList<>();
		l1.add(1233);
		l1.add("test");

		ArrayList<Student<String>> secA=new ArrayList<Student<String>>();
		secA.add(new Student<String>("4ub13ec"));

	}
}
```

Inner class or nested class

Class A

{

Class B{

}

}

Inner class types

CORE JAVA- PART III

1. **Local inner class**
2. **Instance inner class**
3. **Static inner class**
4. **Anonymous inner class**

1. **Local inner class**

```
package pack1;

public class Demo1 {

	int x=12;
	static double y=34.12;
	void test(){
		System.out.println("running test() method.....");
		//Rule 1- no static members
			//final static variable is allowed
		//Rule 2 - scope within the body
		class sample1
		{
			sample1()
			{
				System.out.println("running sample1
constructor......");
			}
			int i=12;
			static final int k=23;
			void disp()
			{
				System.out.println("running disp()......");
			}
		}
		sample1 s1=new sample1();
		System.out.println("i = "+s1.i);
		System.out.println("k = "+sample1.k);
		s1.disp();

	}

}
package pack1;

public class MainClass {
public static void main(String[] args) {

	Demo1 d1=new Demo1();
	d1.test();
}
}
```

```
OUTPUT:
running test() method.....
running sample1 constructor......
i = 12
k = 23
```

```
running disp()......
```

2. Static inner class
   H

```
package pack1;

public class Demo2 {
	int x=12;
	static int y=24;
	void fun1()
	{
		System.out.println("running fun1()......");
	}
	static void fun2() {
		System.out.println("running fun2() ........");
	}
	static class Sample2{
			static int i=78;
				int j=15;
				static void m1()
				{
					System.out.println("Running m1() method...");
				}
				void m2() {
					System.out.println("Running m2() method......");
				}
	}
}
package pack1;

public class MainClass2 {

	public static void main(String[] args) {
	System.out.println(" y value = "+Demo2.y);
	Demo2.fun2();

	Demo2 d1=new Demo2();
	System.out.println("x value is "+d1.x);
	d1.fun1();
	//accessing static members of static inner clas
	System.out.println("i value is "+Demo2.Sample2.i);
	Demo2.Sample2.m1();
	//accessing non static members of static inner class
	Demo2.Sample2 s1=new Demo2.Sample2();
	System.out.println(" j value = "+s1.j);
	s1.m2();

	}
}


OUTPUT:
y value = 24
running fun2() ........
x value is 12
running fun1()......
i value is 78
Running m1() method...
```

```
 j value = 15
Running m2() method......
```

3. Instance inner class
   H

```
package pack1;

public class Demo3 {

	int x=12;
	static int y=24;
	void fun1()
	{
		System.out.println("running fun1()......");
	}
	static void fun2() {
		System.out.println("running fun2() ........");
	}
	//only non static members are allowed
	class Sample3{
			static final int i=78;
				int j=15;

				void m2() {
					System.out.println("Running m2() method......");
				}
	}

}


package pack1;

public class MainClass3 {
	public static void main(String[] args) {
		System.out.println(" y value = "+Demo3.y);
		Demo3.fun2();

		Demo3 d1=new Demo3();
		System.out.println("x value is "+d1.x);
		d1.fun1();
		//accessing static final members of static inner clas
		System.out.println("i value is "+Demo3.Sample3.i);
		//accessing non static members of non-static inner class
		//Demo3.Sample3 s1=new Demo3().new Sample3();
		Demo3.Sample3 s1=d1.new Sample3();
		System.out.println(" j value = "+s1.j);
		s1.m2();

		}

}


OUTPUT:
y value = 24
running fun2() ........
x value is 12
```

```
running fun1()......
i value is 78
 j value = 15
Running m2() method......
```

4. Anonymous inner class

```
package pack1;

public class Demo4 {
	void test()
	{
		System.out.println("defined in Demo4 class");
	}

}
package pack1;

public interface Demo5 {

	void disp();

}
package pack1;

public class MainClass4 {

	public static void main(String[] args) {

		Demo4 d1=new Demo4() {
			void test()
			{
				System.out.println("test() defined in annonympus inner
clsss");
			}

		};
		d1.test();

		Demo5 d2=new Demo5()
		{
			public void disp() {
			System.out.println("disp() defined in annonymous inner
interface");

			}

		};
		d2.disp();

	}
}
OUTPUT:
test() defined in annonympus inner clsss
disp() defined in annonymous inner interface
```

CORE JAVA- PART III

Garbage collection:

Memory management:

- Allocate memory
- Deallocate memory

Garbage collection:

- ➔ New operator allocates the memory(heap)
- ➔ It's a process of cleaning/ freeing up the memory
- ➔ It's built with algorithms

Garbage collector:

- It's a thread, which collects subject and destroy it.
- It's a low priority thread.
- It looks for eligible objects if found, destroy it.
- Dereferenced objects are eligible for garbage collector

Eg. Demo1 d1=new Demo1();

d1=

d1=….

d1=null;

Eg 2: new String("objects").toString().toUpperCase();

- GC should always runs at the end
- Raise a request for GC
- → System.gc
- GC destroys the dereferenced objects
- Before destroying it calls the finalize() method.

final	finally	finalize()
Keyword	Block	It's a non-static method defined objects class

CORE JAVA- PART III

Pass by value:

```
Void m1()
{
Int a=12;
m2(a);
Sop(a);//12 since the value will not be altered, only copy of a will passed to m2()
}
Void m2(int arg)
{
Sop(arg);//12
Arg++;
Sop(arg);//13[
}
```

Pass by reference:

```
Void m1()
{
Int a=12;
m2(a); // passing the address
Sop(a);//13
}
Void m2(int arg) // pointer values
{
Sop(arg);//12
Arg++;
Sop(arg);//13
}
// pass by reference is not supported in java since java doesn't support pointer concepts.
Void m1()
{
Demo1 d1=new Demo1();
```

m2(d1);

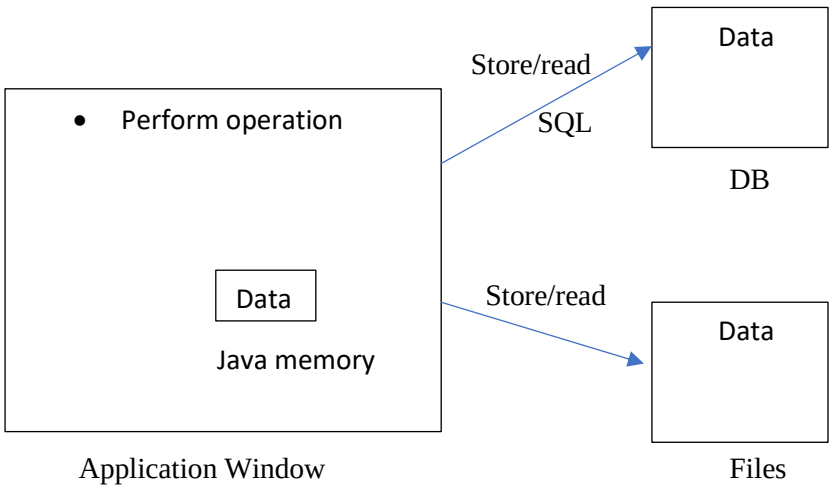
}

Void m2(Demo1 arg)

{

}

File programming



1. File

```
package pack1;

import java.io.File;

/*

* 1. creates an empty file or directory
* delete file or directory
* get the path of the file or directory
* we can get the file permission
* we can set file permission
* File class has overloaded constructor
*/
public class mainclass {
public static void main(String[] args) {

      File f1=new File("E:\\HDB JAVAP\\Music\\Music");
```

```
		f1.mkdir();
		if(f1.exists()) {
			System.out.println("Directory exists");
		}else
		{
			System.out.println("Directory not Exists");
		}
		f1.delete();


}


}
```

- Program to Create a new File

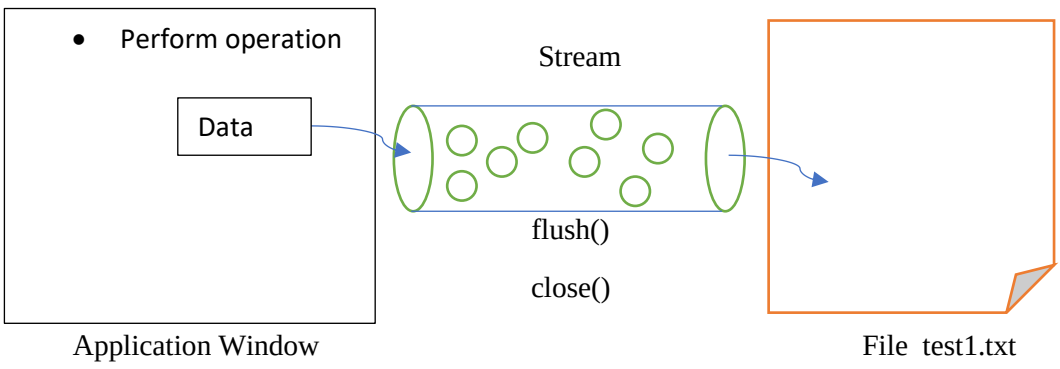
```
package pack1;

import java.io.File;
import java.io.IOException;

public class mainclass2 {
	public static void main(String[] args) {
		String path="E:\\HDB JAVAP\\Music\\Music\\test.txt";
		File f1=new File(path);
		try {
			f1.createNewFile();
		} catch (IOException e) {
			e.printStackTrace();
		}

		if(f1.exists()) {
			System.out.println("File exists");
		}else
		{
			System.out.println("File  does not Exists");
		}
	}

}
```

- Program to write content in a file

```
package pack1;
```

CORE JAVA- PART III

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterDemo {
public static void main(String[] args) {


	String path="E:\\HDB JAVAP\\Music\\Music\\test1.txt";
	File f1=new File(path);
	FileWriter fwrite=null;

	try {
		fwrite=new FileWriter(f1);
		fwrite.write("hiiiiiii hdb \r\n");
		fwrite.write("bye");
		fwrite.flush();

	} catch (IOException e) {
		e.printStackTrace();
	}finally {
		try {
			fwrite.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}




}
}
```



CORE JAVA- PART III

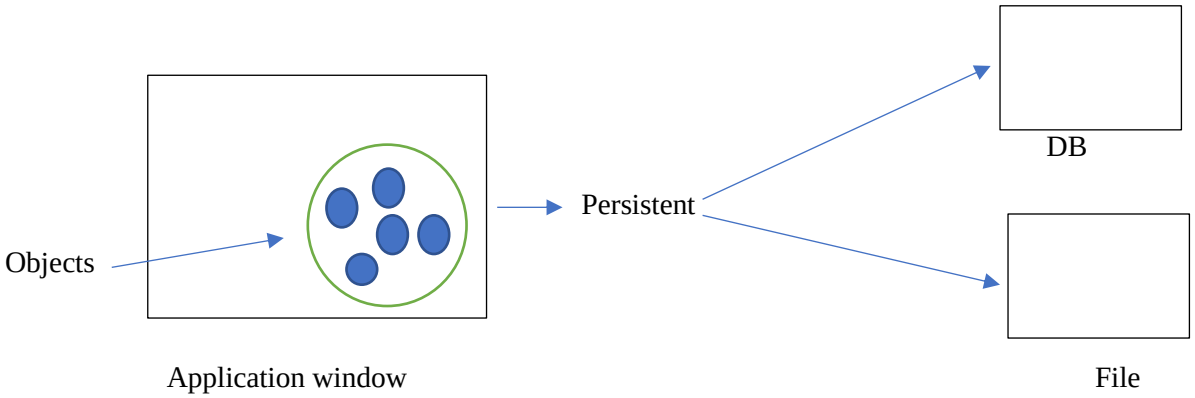
- Program to read file content and print on the console

```
package pack1;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileReaderDemo {


	public static void main(String[] args) throws IOException {

		String path="E:\\HDB JAVAP\\Music\\Music\\test1.txt";
		File f1=new File(path);
		FileWriter fwrite=null;
		int charCount=(int) f1.length();
		FileReader fRead=new FileReader(f1);
		char ch[]=new char[charCount];
		int x=fRead.read();
		 int i=0;
		while(x!=-1)
		{
			ch[i++]=(char) x;
			x=fRead.read();
		}

	String data=new String(ch);
	System.out.println(data);
	fRead.close();

		/*
Another method: just for reading and printing on the console
		  while (x!=-1) {
			System.out.print((char)x);
			x=fRead.read();

		}
		fRead.close();
		*/
	}
}
```

CORE JAVA- PART III

Object Serialization:

- The objects stored in the heap memory are always Non-persistent
- To make objects persistent, we make use of files or DB.
- Saving/writing the objects into the file or databases is called sterilization.



Notes:

- Whenever an object is created in the JVM, the Objects are stored in the heap memory, these objects are non-persistent objects
- They get destroyed or removed from the memory once the JVM stops running.
- In order to make the object persistent, the objects should be saved permanently either in the file or in the database.
- Writing an object into the file is known as serialization.
- serialization can be done only for those objects which is a type of serializable.
- A class which implements serializable interface are eligible for serialization.
- If any member variable is declared as a transient that member variable will not be written into the file.
- Transient keyword should be used only for non-static member variable.
- The objectOutputStream is used for serialization.
- Reading the objects details from a file is known as deserialization
- The deserialization is done by objectInputStream.

- Program to illustrate the Serialization

```
package pack1;

import java.io.Serializable;

public class Employee implements Serializable {
```

```
	int id;
	String name;
	double salary;
	public Employee(int id, String name, double salary) {
		this.id = id;
		this.name = name;
		this.salary = salary;
	}


}
```

```
package pack1;

import java.io.File;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;


public class MainClass_Ser {

	public static void main(String[] args) throws Exception {
		System.out.println("main method started");

		File f1=new File("E:\\HDB
JAVAP\\Music\\Music\\Employee.ser");

		FileOutputStream fout=new FileOutputStream(f1);
		ObjectOutputStream fobj=new ObjectOutputStream(fout);

		Employee e1=new Employee(1234, "hdb", 34000.00);

		fobj.writeObject(e1);
		fobj.flush();
		fobj.close();
		System.out.println("main method ended");



	}

}
```

```
package pack1;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
import java.io.ObjectInputStream;

public class MainClass_deSer {

	public static void main(String[] args) throws Exception {

		System.out.println("main  method started");
		File f1=new File("E:\\HDB
JAVAP\\Music\\Music\\Employee.ser");

		FileInputStream fin=new FileInputStream(f1);
		ObjectInputStream fobj=new ObjectInputStream(fin);
		Employee e1=(Employee) fobj.readObject();

		System.out.println("Employee Id: "+e1.id);
		System.out.println("Employee name: "+e1.name);
		System.out.println("Employee salary: "+e1.salary);

		fobj.close();
		System.out.println("main method ended");
	}

}
```

```
OUTPUT:
main  method started
Employee Id: 1234
Employee name: hdb
Employee salary: 34000.0
main method ended
```

CORE JAVA- PART III