- **Comparable Interface**:
  The `Comparable` interface is used to define the **natural ordering** of objects. It allows an object to compare itself with another object of the same type. The interface has one method:

  public int compareTo(T o);

  Return Value:

  Negative: If the current object is less than the other object.
  Zero: If the current object is equal to the other object.
  Positive: If the current object is greater than the other object.

Code Explanation

In your code, the `Student` class implements `Comparable<Student>`:

java

```
class Student implements Comparable<Student> {
   int rollno;
   String name;

   Student(int rollno, String name) {
      this.rollno = rollno;
      this.name = name;
   }

   @Override
   public int compareTo(Student other) {
      return this.rollno - other.rollno; // Natural ordering based on rollno
   }

   @Override
   public String toString() {
      return "[" + rollno + ", " + name + "]";
   }
}
```

Natural Ordering: The `compareTo` method defines the natural ordering of `Student` objects based on their `rollno`. This means that when you sort a list of `Student` objects, they will be ordered by their `rollno`.

Usage Example

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
```

```
        List<Student> students = new ArrayList<>();
        students.add(new Student(3, "Charlie"));
        students.add(new Student(1, "Alice"));
        students.add(new Student(2, "Bob"));

        // Sorting using natural ordering (defined by compareTo)
        Collections.sort(students);

        System.out.println(students); // Output: [[1, Alice], [2, Bob], [3, Charlie]]
    }
}
```

Key Points:

- The `Collections.sort()` method uses the `compareTo` method defined in the `Comparable` interface to sort the list.

- The `toString()` method is overridden to provide a readable representation of the `Student` objects.

- **Comparator Interface**
  The `Comparator` interface is used to define **custom ordering** for objects. Unlike `Comparable`, which defines the natural ordering, `Comparator` allows you to define multiple sorting criteria for the same class.

  The interface has one key method:

  public int compare(T o1, T o2);

Return Value:

- Negative: If `o1` is less than `o2`.
- Zero: If `o1` is equal to `o2`.
- Positive: If `o1` is greater than `o2`.

Why Use Comparator?*

- When you want to sort objects in a way that differs from their natural ordering.

- When you cannot modify the original class (e.g., if it's part of a library).

Example Using Comparator

You can create a `Comparator` to sort `Student` objects by `name` instead of `rollno`:

```
import java.util.*;
```

```
public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(3, "Charlie"));
        students.add(new Student(1, "Alice"));
        students.add(new Student(2, "Bob"));

        // Sorting using a custom Comparator (by name)
        Comparator<Student> nameComparator = new Comparator<Student>() {
            @Override
            public int compare(Student s1, Student s2) {
                return s1.name.compareTo(s2.name); // Compare names lexicographically
            }
        };
        Collections.sort(students, nameComparator);
        System.out.println(students); // Output: [[1, Alice], [2, Bob], [3, Charlie]]
    }
}
```

Differences Between `Comparable` and `Comparator`

Feature	`Comparable`	`Comparator`
Purpose	Defines the natural ordering of a class.	Defines custom ordering for a class.
Method	`compareTo(T o)`	`compare(T o1, T o2)`
Location	Implemented in the class itself.	Implemented externally (as a separate class or lambda).
Flexibility	Limited to one natural ordering.	Allows multiple sorting criteria.
Use Case	When the class controls its own ordering.	When you need additional sorting logic.

Ways to Use `Comparator`

a. Anonymous Class

You can define a `Comparator` using an anonymous class:

```
Comparator<Student> nameComparator = new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
};
```

Lambda Expression (Java 8+)

Lambda expressions simplify the creation of `Comparator` instances:

```
Comparator<Student> nameComparator = (s1, s2) -> s1.name.compareTo(s2.name);
```

Method References

If the comparison logic is already defined in a method, you can use a method reference:

```
Comparator<Student> nameComparator = Comparator.comparing(s -> s.name);
```

Or even shorter:

```
Comparator<Student> nameComparator = Comparator.comparing(Student::getName);
```

Chained Comparators

You can chain multiple comparators to define complex sorting logic:

```
Comparator<Student> rollnoThenNameComparator = Comparator
    .comparingInt(Student::getRollno)
    .thenComparing(Student::getName);
```

Important Things to Know

a. Stability of Sorting

- Sorting algorithms like `Collections.sort()` are **stable**, meaning that equal elements retain their relative order.

b. Null Handling

- Both `Comparable` and `Comparator` can throw `NullPointerException` if `null` values are compared.

- To handle `null` values, you can use `Comparator.nullsFirst()` or `Comparator.nullsLast()`:

Comparator<Student> nullSafeComparator = Comparator.comparing(Student::getName, Comparator.nullsFirst(Comparator.naturalOrder()));

c. Reverse Order

- You can reverse the order of a `Comparator` using `reversed()`:

Comparator<Student> reverseNameComparator = Comparator.comparing(Student::getName).reversed();

d. Primitive-Specific Comparators

- For primitive types, use specialized comparators like `Comparator.comparingInt()`, `Comparator.comparingDouble()`, etc., for better performance.

Comparator<Student> rollnoComparator = Comparator.comparingInt(Student::getRollno)

Summary of Usage Patterns

Pattern	Example
Comparable (Natural)	`Collections.sort(students);`
Comparator (Anonymous)	`Collections.sort(students, new Comparator<Student>() { ... });`
Comparator (Lambda)	`Collections.sort(students, (s1, s2) -> s1.name.compareTo(s2.name));`
Comparator (Method Ref)	`Collections.sort(students, Comparator.comparing(Student::getName));`
Chained Comparators	`Comparator.comparingInt(Student::getRollno).thenComparing(Student::getName);`
Null-Safe Comparator	`Comparator.comparing(Student::getName, Comparator.nullsFirst(...));`
Reverse Order	`Comparator.comparing(Student::getName).reversed();`

Final Answer

- Comparable is used to define the **natural ordering** of objects within the class itself.

- Comparator is used to define **custom ordering** externally, allowing flexibility for multiple sorting criteria.

- Both can be implemented using traditional methods, anonymous classes, lambda expressions, or method references.

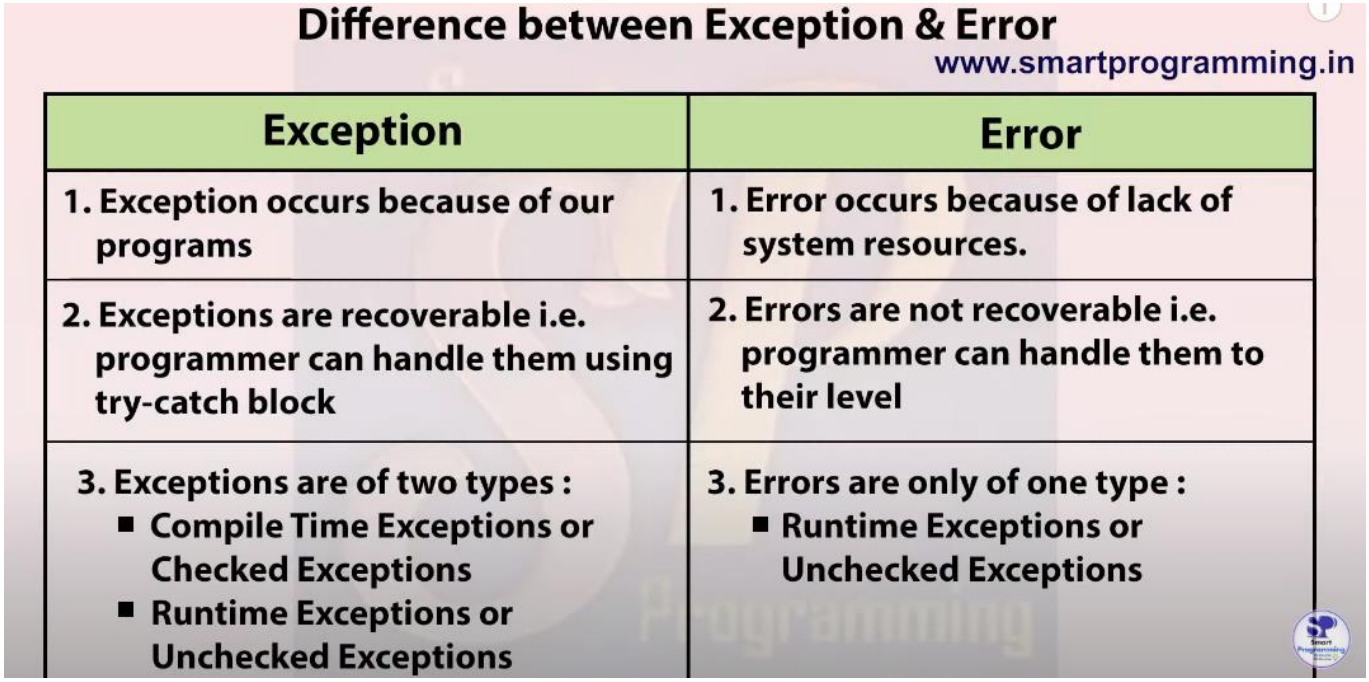
- Key features include handling `null` values, reversing order, chaining comparators, and optimizing for primitives.

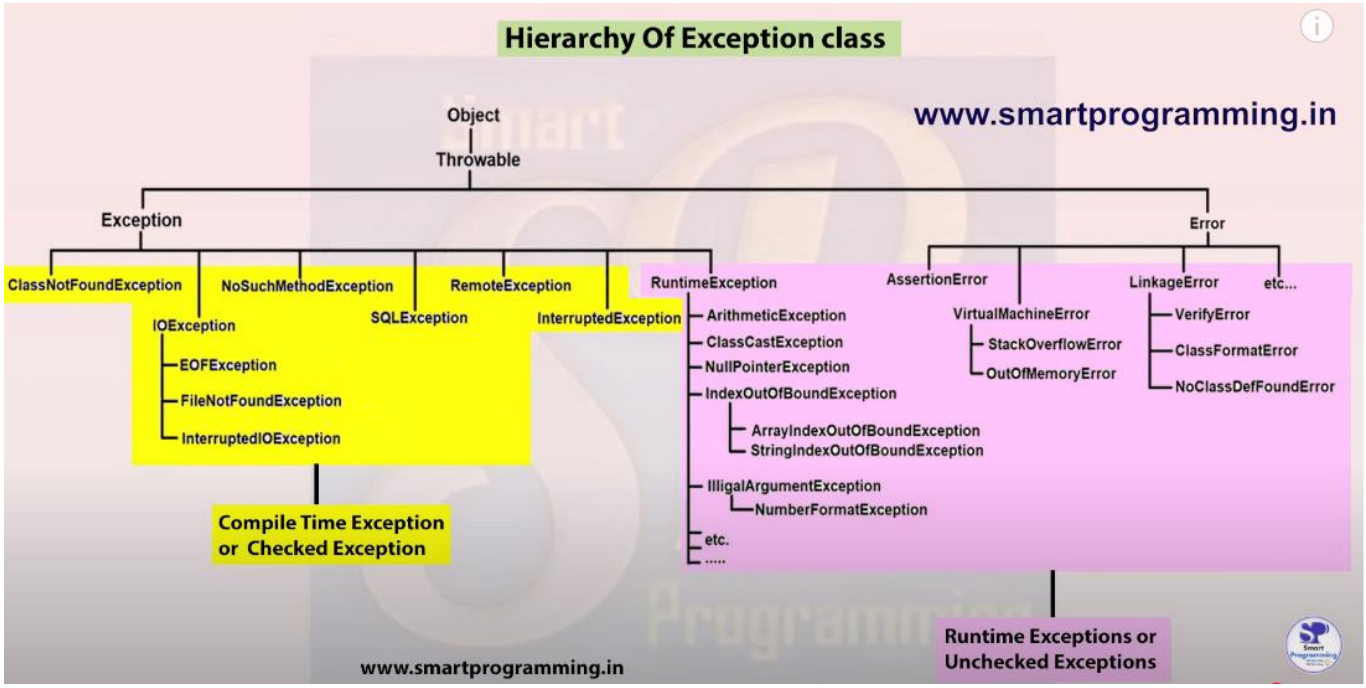
By understanding and leveraging both `Comparable` and `Comparator`, you can write flexible and efficient sorting logic in Java.

Exception Handling

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time, that disrupts the normal flow of the program.

When such unexpected event and we solve it with alternate way and we called it as Exception handling





Exceptions always occur at runtime. **Exception does not occur at compile time.**

Compile time pe compiler warn karta hai ki future mai ye exception aa sakti hai isse handle karo (using try-catch) aur ye exception nhi hota. Jab ham warning ko handle kar lete hai tab code compile hoga but jo warning mili thi waise hi problem hai to runtime pe exceotion aaegi.

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis=new FileInputStream("d:/abc.txt");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Abc.txt does not exists.

```
D:\>cd "java programs"

D:\java programs>javac Test.java
Test.java:7: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
                FileInputStream fis=new FileInputStream("d:/abc.txt");
                                  ^
1 error

D:\java programs>javac Test.java

D:\java programs>java Test
java.io.FileNotFoundException: d:\abc.txt (The system cannot find the file specified)
```

1st compiler bata raha hai warn kar rha hai

2nd exception occur hui

Checked Exception (**Compile** time exception)

Aisi exception jo compiler apne level pe check kar sakta hai aur bolta hai isse handle karo (try-cath, throw, throws).

```
//FileInputStream fis=new FileInputStream("d:/abc.txt");
//Class.forName("com.mysql.jdbc.Driver");
```

Unchecked Exception (**Runtime** exception)

In unchecked compiler is not able to check the exception. Compiler Ignore it.

```
int a=100, b=0, c;
c = a/b;
System.out.println(c);
```

Checked Exception / Compile Time Exception	Unchecked Exception / Runtime Exception
1. Checked Exceptions are the exceptions that are checked and handled at compile time.	1. Unchecked Exceptions are the exceptions that are not checked at compiled time.
2. The program gives a compilation error if a method throws a checked exception.	2. The program compiles fine because the compiler is not able to check the exception.
3. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.	3. A method is not forced by compiler to declare the unchecked exceptions thrown by its implementation. Generally, such methods almost always do not declare them, as well.
4. A checked exceptions occur when the chances of failure are too high.	4. Unchecked exception occurs mostly due to programming mistakes.
5. They are direct subclass of Exception class but do not inherit from RuntimeException.	5. They are direct subclass of RuntimeException class.

How to handle exception (Checked & Unchecked).

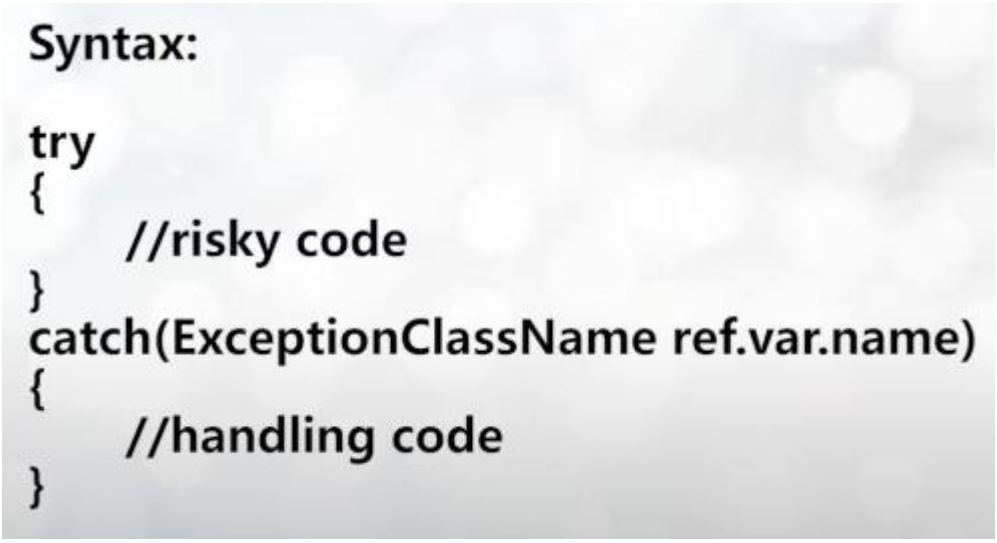
Whenever there is exception, the method in which exception occurs will create an object and that object will store three things.

1. Exception name
2. Description
3. Stack trace (Proper long description)

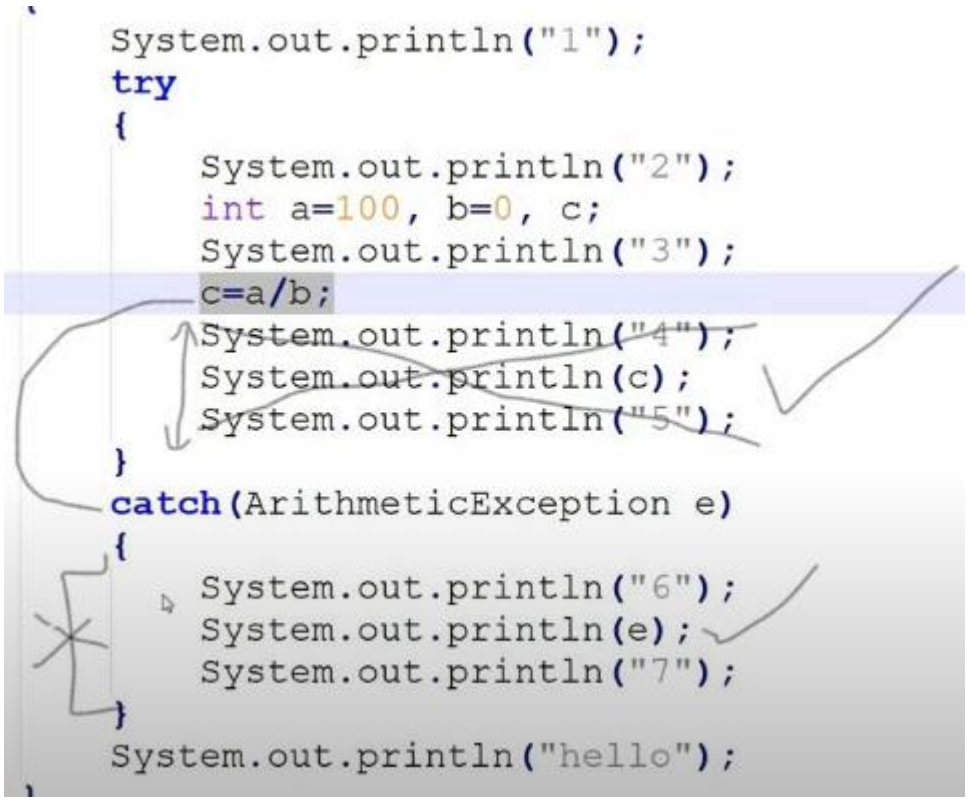
JVM Default Exception Handler (method) ye object pass kar dega agar use handle nhi kiya gya hai aur use pehle program abnormally terminate ho jaega.

We can handle the exception using 5 keywords

1. Try
2. Catch
3. Finally
4. Throw
5. Throws



Flow of try-catch



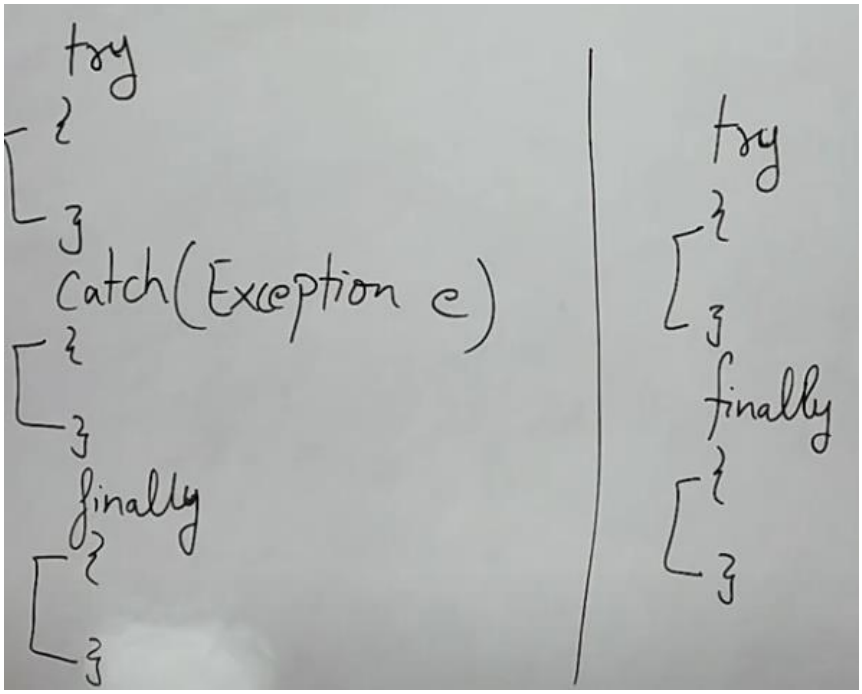
```
System.out.println("1");
try
{
    System.out.println("2");
    int a=100, b=0, c;
    System.out.println("3");
    c=a/b;
    System.out.println("4");
    System.out.println(c);
    System.out.println("5");
}
catch(ArithmeticException e)
{
    System.out.println("6");
    System.out.println(e);
    System.out.println("7");
}
System.out.println("hello");
```

Printing Exception

1. E.printStackTrace() (Most Used)
   Name, description, stack trace (line)
2. Sout(e)
   Name, description
3. Sout(e.getMessage())
   description

Finally Block:

Always executes whether exception occurred or not. It is used to write a **clean up code** like closing db connection, resources that we tried to open in try we will close that in finally block.



Try ke baad directly finally use kar sakte hai but exception handle nhi hoga. Preferred way to use is try-catch-finally

```java
ArrayList<Integer> list = new ArrayList<>();
list.add(311231);
list.add(14130);
list.add(11121);
list.add(5239);
list.add(41232);

// System.out.println(list);
Collections.sort(list, (a,b)->(a%10-b%10)==0 ? a-b:a%10-b%10
    );
System.out.println(list);


try{
    System.out.println(1/0);
}
// catch(Exception e){
//     System.out.println("Exception occured");
// }
finally{
    System.out.println("Inside Finally");
}

System.out.println("Hii hello");
```

```
[14130, 11121, 311231, 41232, 5239]
Inside Finally
ERROR!
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:22)

=== Code Exited With Errors ===
```

Akela try, catch or finally block nhi allowed hai compilation error

```
try{
    System.out.println(1/0);
}
catch(Exception e){
    System.out.println("Exception occured");
    return;
}
finally{
    System.out.println("Inside Finally");
}

System.out.println("Hii hello");
```

```
[14130, 11121, 311231, 41232, 5239]
Exception occured
Inside Finally
```

If return is in try or catch block than also finally will run but below finally codes will not run

```
public static void main(String[] args) throws IOException
{
    FileInputStream fis = null;
    try
    {
        fis=new FileInputStream("d:/abc.txt");
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File not found");
    }
    finally
    {
        // cleanup here
        if (fis!=null)
        {
            fis.close();
        }
        System.out.println("file closed");
    }
```

We can use multiple catch blocks with one try block but we can only use single finally block with one try block, not multiple.

The statements present in the finally block execute even if the try block contains control transfer statements (i.e. jump statements) like ***return, break or continue.***

Four Scenarios where finally block does not execute

1. Using of System.exit(0) method inside try
2. Causing a fatal error that causes the process to abort (outofmememory etc.)
3. Due to exception arising in the finally block
4. The death of a Thread

Difference between **final, finally, finalize**

Finalize method is just executed prior to garbage collector. Clean is performed

```
class Main {
    public static void main(String[] args) {
        for(int i=0; i<300; i++){
            A a = new A();
            A.count++;
        }
    }
}

class A{
    static int count=0;
    A(){
        System.out.println("Object created "+count);
    }
    protected void finalize() throws Throwable{
        System.out.println("Finalize: Object of A is being destoryed!"
            );
    }
}
```

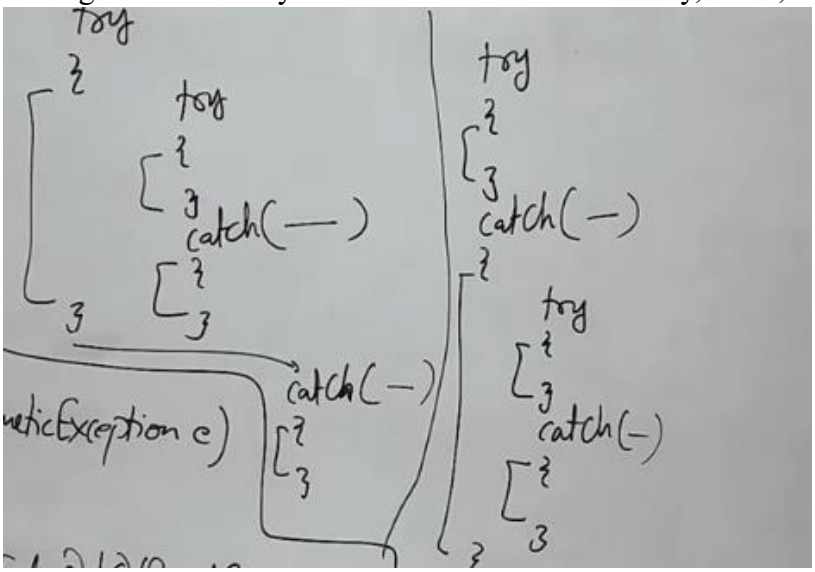
Possible combinations of try-catch-finally

1. Alone any cannot exist
2. Multiple catch block can be used with single try but Parent (Exception) should be last. If it is first than error. Two same classes in catch cannot be provided

```
try{
    System.out.println(1/0);
}
catch(Exception e){

}
catch(ArithmeticException e){

}
```

```
/tmp/4Nb22xmxDi/Main.java:18: error: exception ArithmeticException has alread
    caught
        catch(ArithmeticException e){
        ^
```

3. Nesting is allowed. Try-Catch can be written inside try, catch, and finally block



4. Try ke baad finally directly

Throw:

Throw keyword is used to throw user defined exceptions. If you use existing exception the program will run fine but that is not recommended. Customized exception ke liye throw keyword use hota hai.

Here a programmer manully creates a exception object using throw keyword. But exception handle karna padega using try-catch kyunki throw bhi abnormally program ko terminate kar dega

throw new UserException();

Check exception when extending to Exception class

```
class YoungerAgeException extends Exception
{

}
```

Uncheck

```
class YoungerAgeException extends RuntimeException
{

}
```

It is better to make customized exception unchecked

```java
import java.util.Scanner;

class YoungerAgeException extends RuntimeException
{
    YoungerAgeException(String msg)
    {
        super(msg);
    }
}
```

```java
class Voting
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("Enter Your Age : ");

        int age=s.nextInt();
        if(age<18)
        {
            throw new YoungerAgeException("You are not eligible for voting");
        }
        else
        {
            System.out.println("you can vote successfully");
        }
        System.out.println("hello");
    }
}
```

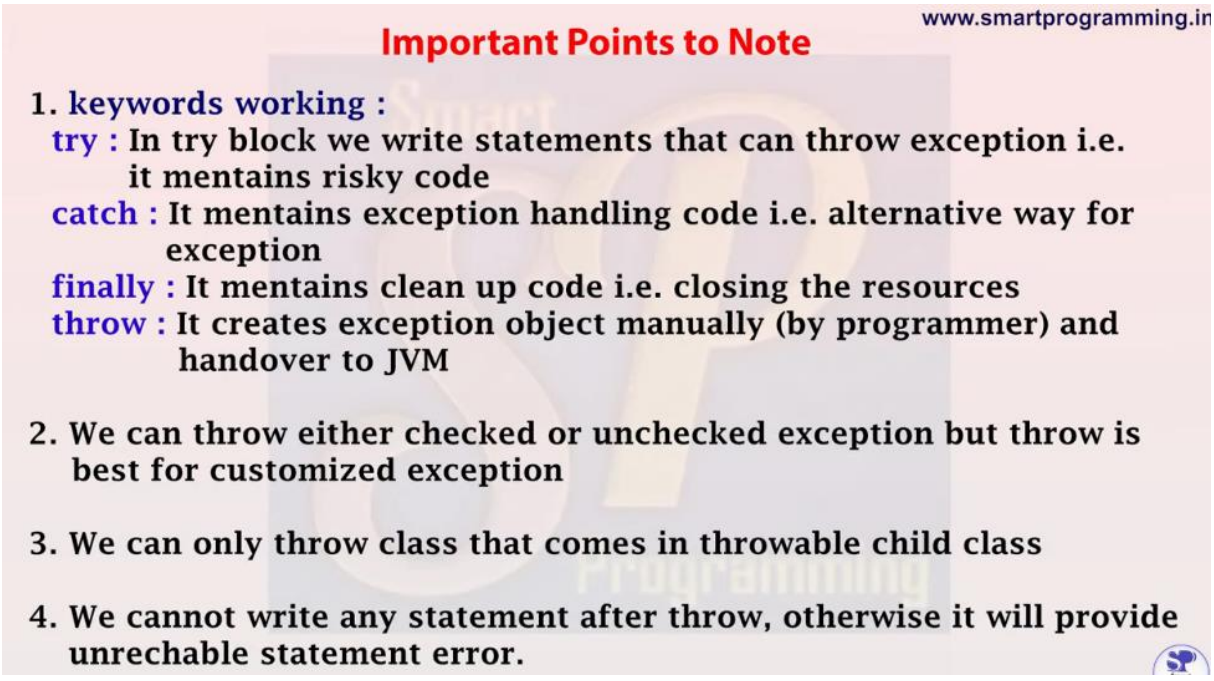
User made a custom exception and had use throw keyword to manually create a exception object. JVM ask main method did you handle this exception main said No than JVM gives the exception object to Default Exception handler DEH and it prints the exception object and program ko abnormally terminate kar dega.

Handling throw exception

```
Scanner s=new Scanner(System.in);
System.out.println("Enter Your Age : ");

int age=s.nextInt();
try
{
    if(age<18)
    {
        throw new YoungerAgeException("You are not eligible for voting");
    }
    else
    {
        System.out.println("you can vote successfully");
    }
}
catch(YoungerAgeException e)
{
    e.printStackTrace();
}
System.out.println("hello");
```

Summary

1. Throw is used to throw user defined exception that are mainly unchecked (RuntimeException)
2. It manually create an exception object and gives it to jvm
3. You cannot write any statement after throw will give compilation error

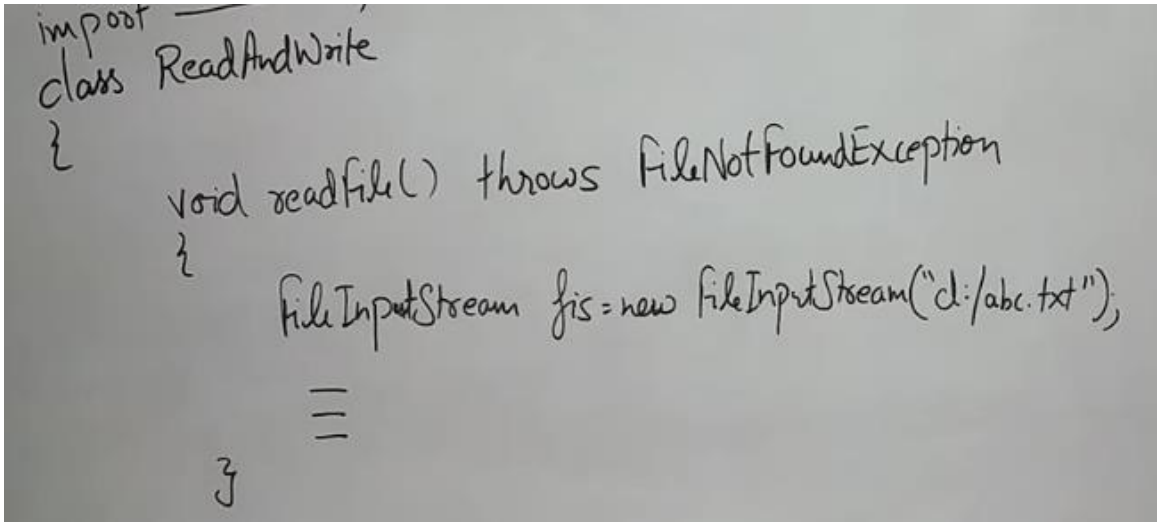


Throws **(**method ke sath use hot hai**)**

Throws keyword is used to declare an exception. It gives an information to the caller method that there may occur an exception so it is better for the caller method to provide the exception handling code so that normal flow can be maintained.

Ye maine ek method bana diya isse tum use karo but be cautious as this method can throw an exception so please handle it.

FileInputStream class throws "FileNotFoundException" which is compile time exception or checked exception so we have to handle the exception and for this purpose we have to use either try-catch or throws keyword



"throws" keyword is used to declare an exception. It gives an information to the caller method that there may occur an exception so it is better for the caller method to provide the exception handling code so that normal flow can be maintained.

Throws is used with checked exception

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

class ReadAndWrite
{
    void readFile() throws FileNotFoundException
    {
        FileInputStream fis=new FileInputStream("d:/abc.txt");
        //statements
    }
    void saveFile() throws FileNotFoundException
    {
        String text="this is demo";
        FileOutputStream fos=new FileOutputStream("d:/xyz.txt");
        //statements
    }
}
```

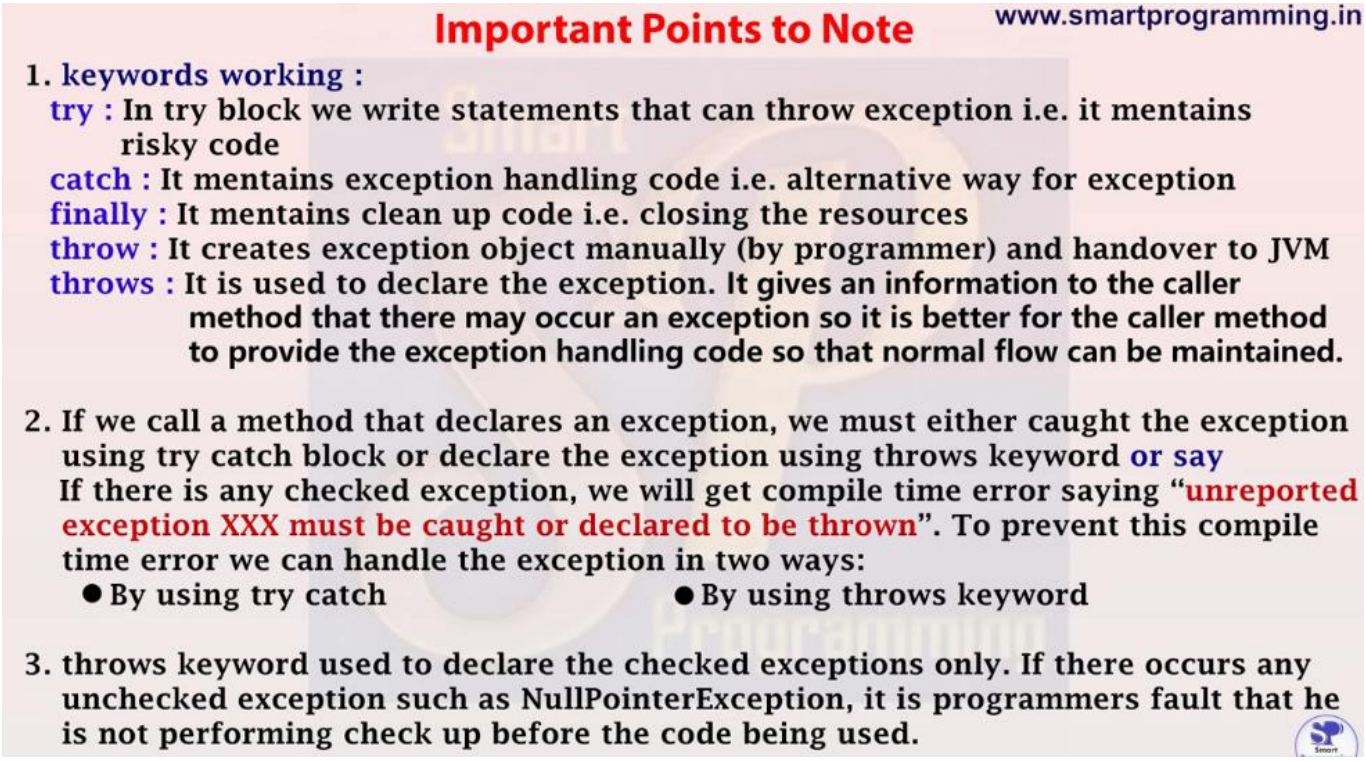
Handling exception

```
class Test
{
    public static void main(String[] args)
    {
        ReadAndWrite rw=new ReadAndWrite();
        try
        {
            rw.readFile();
        }
        catch(FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("hello");
    }
}
```

Compiled but will not handle exception and program will terminate abnormally.

```
class Test
{
    public static void main(String[] args) throws FileNotFoundException
    {
        ReadAndWrite rw=new ReadAndWrite();
        rw.readFile();
        System.out.println("hello");
    }
}
```

throws keyword is used to declare only for the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.



Difference between throw and throws

Difference between throw and throws keyword

throw keyword	throws keyword
1. throw keyword is used to create an exception object manually i.e. by programmer (otherwise by default method is responsible to create exception object)	1. throws keyword is used to declare the exceptions i.e. it indicate the caller method that given type of exception can occur so you have to handle it while calling.
2. throw keyword is mainly used for runtime exceptions or unchecked exceptions	2. throws keyword is mainly used for compile time exceptions or checked exceptions
3. In case of throw keyword we can throw only single exception	3. In case of throws keyword we can declare multiple exceptions i.e. void readFile() throws FileNotFoundException, NullPointerException, etc.
4. throw keyword is used within the method	4. throws keyword is used with method signature
5. throw keyword is followed by new instance	5. throws keyword is followed by class
6. We cannot write any statement after throw keyword and thus it can be used to break the statement	6. throws keyword does not have any such rule

Custom Exception

Checked exception so extend Exception class

If throwing an checked exception so use try-catch or throws to compile the code.

```java
// Online Java Compiler
// Use this editor to write, compile and run your Java code online
import java.util.*;
class Main {
    public static void main(String[] args) throws UnderAgeException{

        int age=17;
        if(age<17){
            throw new UnderAgeException();
        }
        System.out.println("Hii");
    }
}

class UnderAgeException extends Exception{
    UnderAgeException(){
        super("Below 18");
    }
    UnderAgeException(String msg){
        super(msg);
    }
}
```

Unchecked exception so extend RuntimeException class

The code will **compile** successfully without try-catch or throws but during runtime it can give exception

```
public static void main(String[] args)
{
    int age=16;
    try
    {
        if(age<18)
        {
            throw new UderAgeException("You cannot vote as your age is below 18")
        }
        else
        {
            System.out.println("you can vote now...!!");
        }
    }
    catch(UderAgeException e)
    {
        e.printStackTrace();
    }
    System.out.println("hello");
}
```

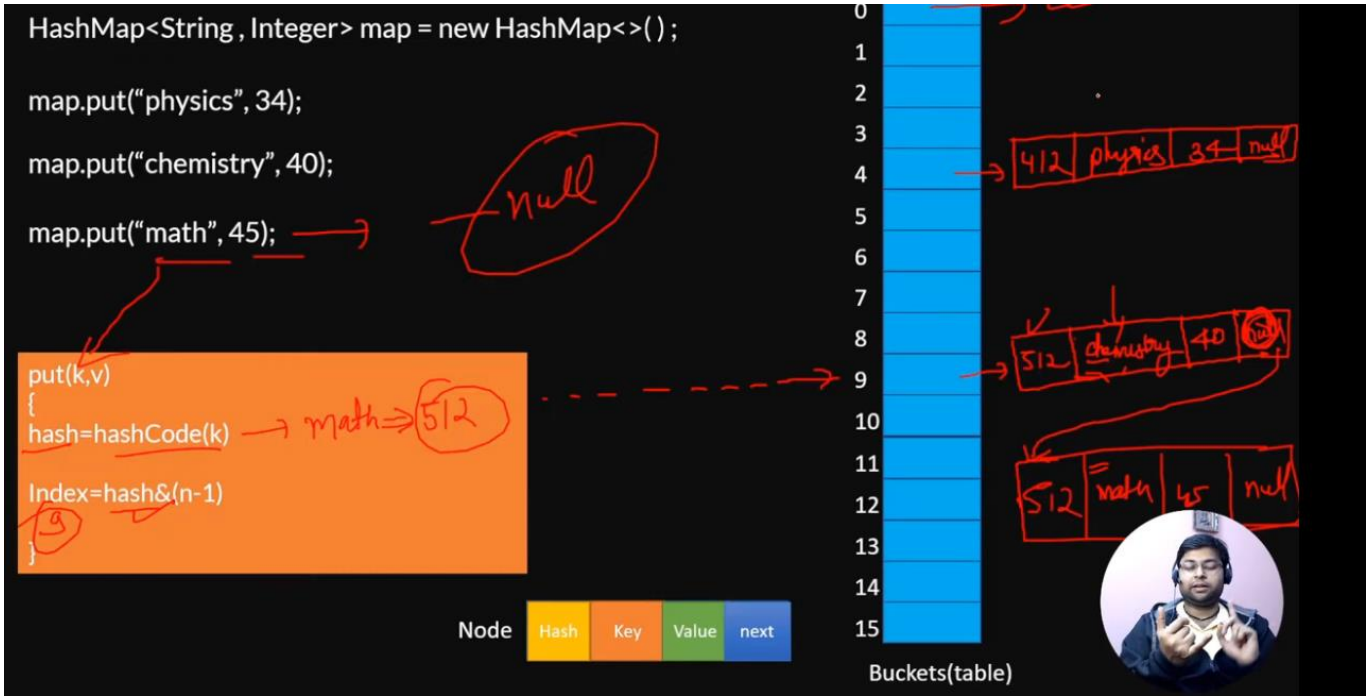
Throw can be used to throw system exception as well as customized exception. But best for throwing customized exception.

Working of HashMap.

https://www.youtube.com/watch?v=sw-j_ETGBEo

Put(null,null) get stored at 0^{th} index.

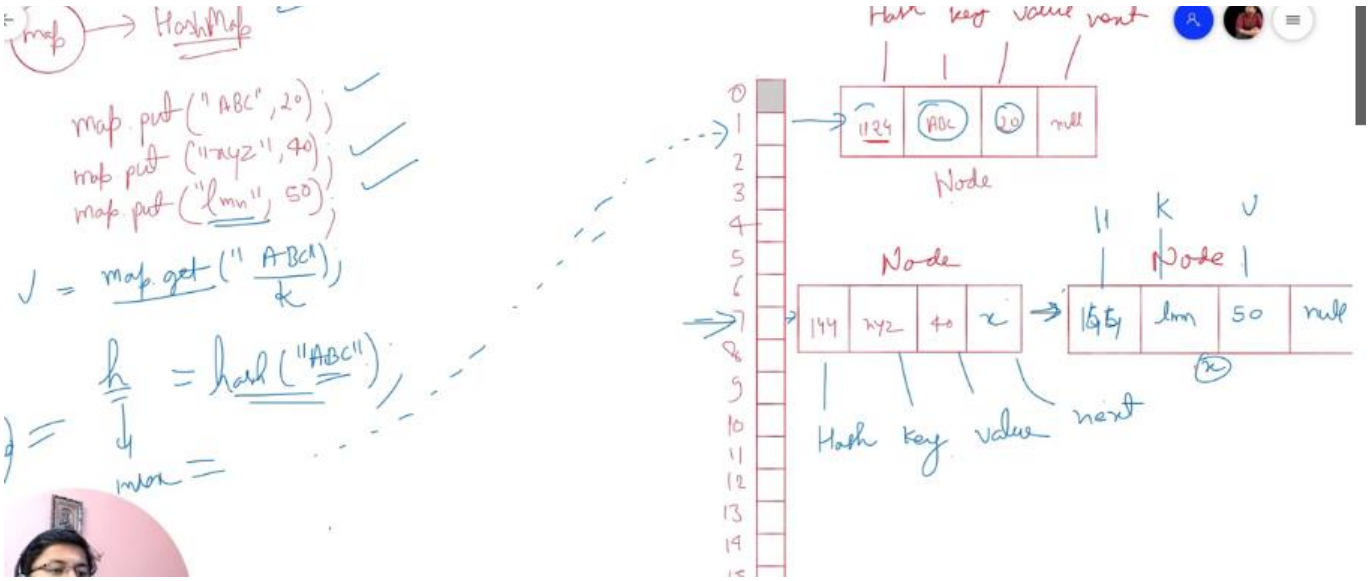
In form of ll node key, values are stored if collision occurs than chaining



From java 8 onward if there is huge collision and size of a particular bucket increases the threshold than BST is used to reduced the TC of put and get from O(n) to O(logn)

Get Operation:

https://www.youtube.com/watch?v=X2k-NTmswVs



Top 15 Hashmap interview questions.

https://www.youtube.com/watch?v=OGGHFabY9G4&list=PL0zysOflRCen9SPmMO2XN1I2S9m96G_dx&index=6&t=18s

HashMap can have **one null key** and any number of null values.

Q) Which data structure HashMap represents ?

A) **Hash Table** data structure

O(1) time if we have the key.

Q) Which data structure is used to implement HashMap in Java?

A) **Array and LinkedList**

For buckets

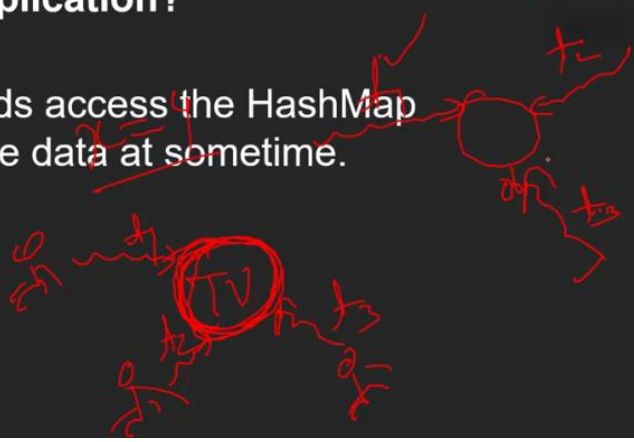
Mappings which land in the same bucket.

Q) Is HashMap thread-safe in Java?

A) No , HashMap is not thread safe

Q) What will happen if you use HashMap in a multithreaded Java application?

A) When multiple threads access the HashMap then they can modify the data at sometime.



Use ConcurrentHashMap for thread safe

Trying to **remove** key from map while iterating not possible

```
HashMap<Integer,Integer> map=new HashMap<>();
map.put(3,11);
map.put(2,10);
for(Map.Entry<Integer,Integer> entry:map.entrySet()){
    if(entry.getKey()==2) map.remove(2);
}
```

```
HashMap<Integer,Integer> map=new HashMap<>();
map.put(3,11);
map.put(2,10);
for(Integer key:map.keySet()){
    if(key==2) map.remove(key);
}
```

```
ERROR!
Exception in thread "main" java.util.ConcurrentModificationException
    at java.base/java.util.HashMap$HashIterator.nextNode(HashMap.java:1605)
    at java.base/java.util.HashMap$EntryIterator.next(HashMap.java:1638)
    at java.base/java.util.HashMap$EntryIterator.next(HashMap.java:1636)
    at Main.main(Main.java:17)
```

We can use iterator on keyset() and keyentry() to remove while iterating.

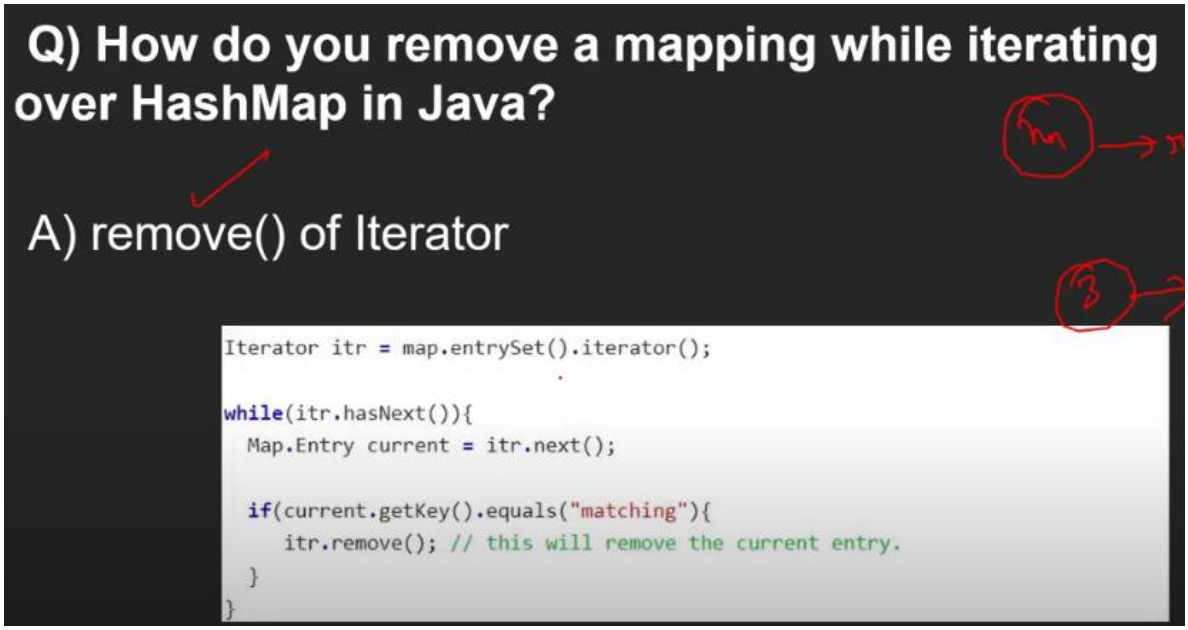
```java
    public static void main(String[] args) {
        HashMap<Integer,Integer> map=new HashMap<>();
        map.put(key:3,value:11);
        map.put(key:2,value:10);

        // ConcurrentModificationExceptio
        for(Integer key:map.keySet()){
            if(key==2) map.remove(key);
        }

        // ConcurrentModificationExceptio
        for(Map.Entry<Integer,Integer> entry:map.entrySet()){
            if(entry.getKey()==2) map.remove(key:2);
        }

        // allowed
        System.out.println(map);
        Iterator itr = map.keySet().iterator();
        while(itr.hasNext()){
            Integer current = (Integer)itr.next();
            if(current==2){
                itr.remove();
            }
        }
        System.out.println(map);

        // Use the entrySet() iterator to iterate over key-value pairs
        Iterator<Map.Entry<Integer, Integer>> itr = map.entrySet().iterator();
        while (itr.hasNext()) {
            Map.Entry<Integer, Integer> current = itr.next();
            if (current.getKey() == 2) {
               itr.remove(); // Safely remove the entry
            }
        }
        System.out.println(map);
    }
```

b. Using `forEach` with `removeIf` (Java 8+)

If you're using Java 8 or later, you can use the `removeIf` method on the `keySet()` or `entrySet()`

```java
map.entrySet().removeIf(entry -> entry.getKey() == 2);
System.out.println(map); // Output: {1=One, 3=Three}
```

- **Explanation** :
  - `removeIf` is a concise way to remove elements based on a condition.
  - It avoids the need for an explicit iterator.



removeIf() can also be used.

6. Important Concepts to Understand

a. Fail-Fast vs. Fail-Safe Iterators

- **Fail-Fast Iterators** :
  - Throw a `ConcurrentModificationException` if the collection is modified during iteration (e.g., `HashMap` iterators).
  - Examples: Iterators returned by `HashMap` , `ArrayList` , etc.
- **Fail-Safe Iterators** :
  - Do not throw a `ConcurrentModificationException` because they operate on a copy of the collection.
  - Examples: Iterators in `ConcurrentHashMap` , `CopyOnWriteArrayList` .

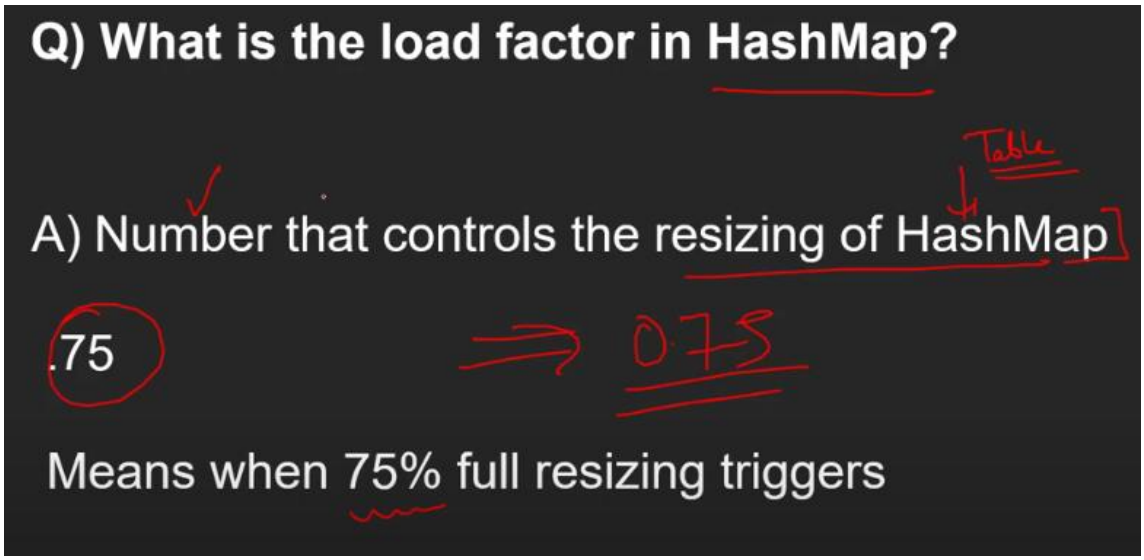
b. `Iterator` Methods

- `hasNext()` : Checks if there are more elements.
- `next()` : Returns the next element.
- `remove()` : Removes the last element returned by `next()` .

Set:

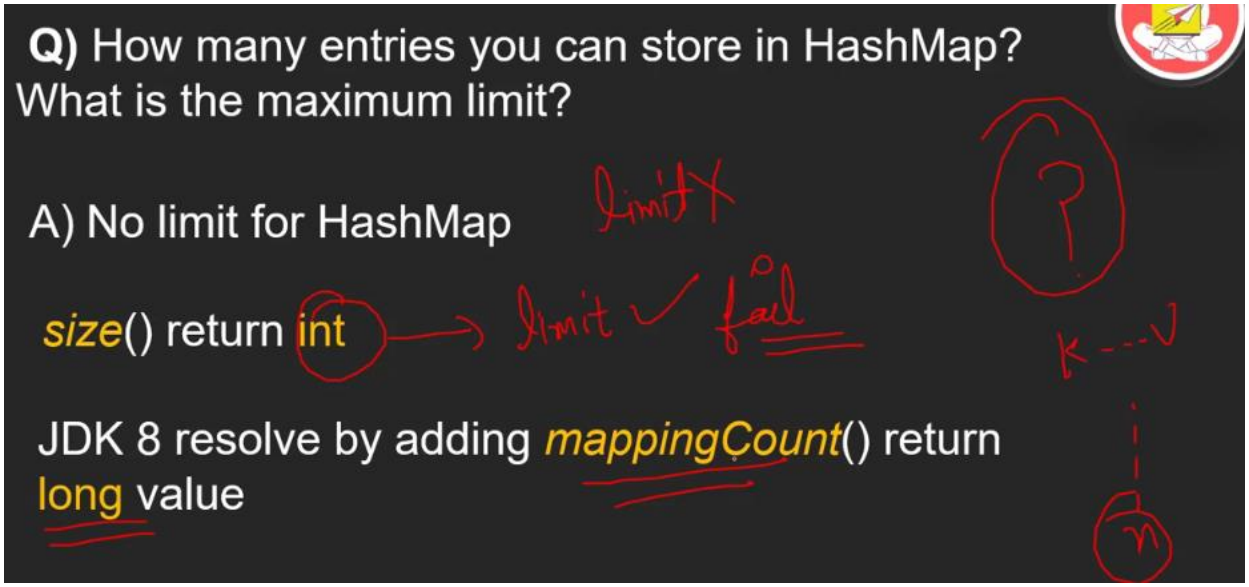
The same concept applies to **Set**. You cannot modify a **Set** directly while iterating over it using a **for-each** loop or other means. Instead, you must use the **Iterator's remove() method** or the **removeIf method** in Java 8+. Understanding these concepts is crucial for working with Java collections and answering related interview questions effectively.

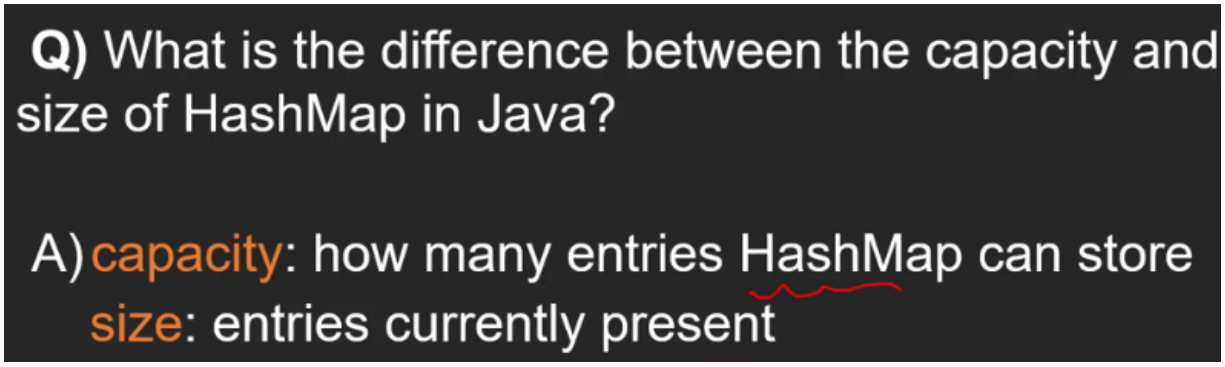
```java
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class SetIterationRemoval {
    public static void main(String[] args) {
        Set<Integer> numbers = new HashSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        numbers.add(5);

        Iterator<Integer> iterator = numbers.iterator();
        while (iterator.hasNext()) {
            Integer number = iterator.next();
            if (number % 2 == 0) { // Remove even numbers
                iterator.remove();
            }
        }
        System.out.println(numbers); // Output: [1, 3, 5]
    }
}
```

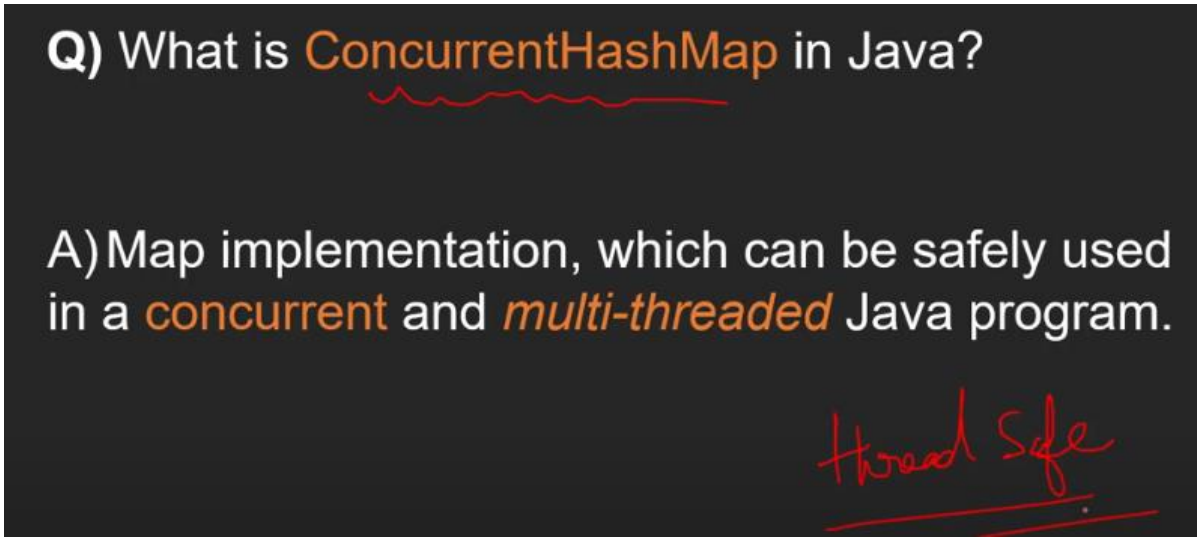


0.75f load factor

Intial capacity 16

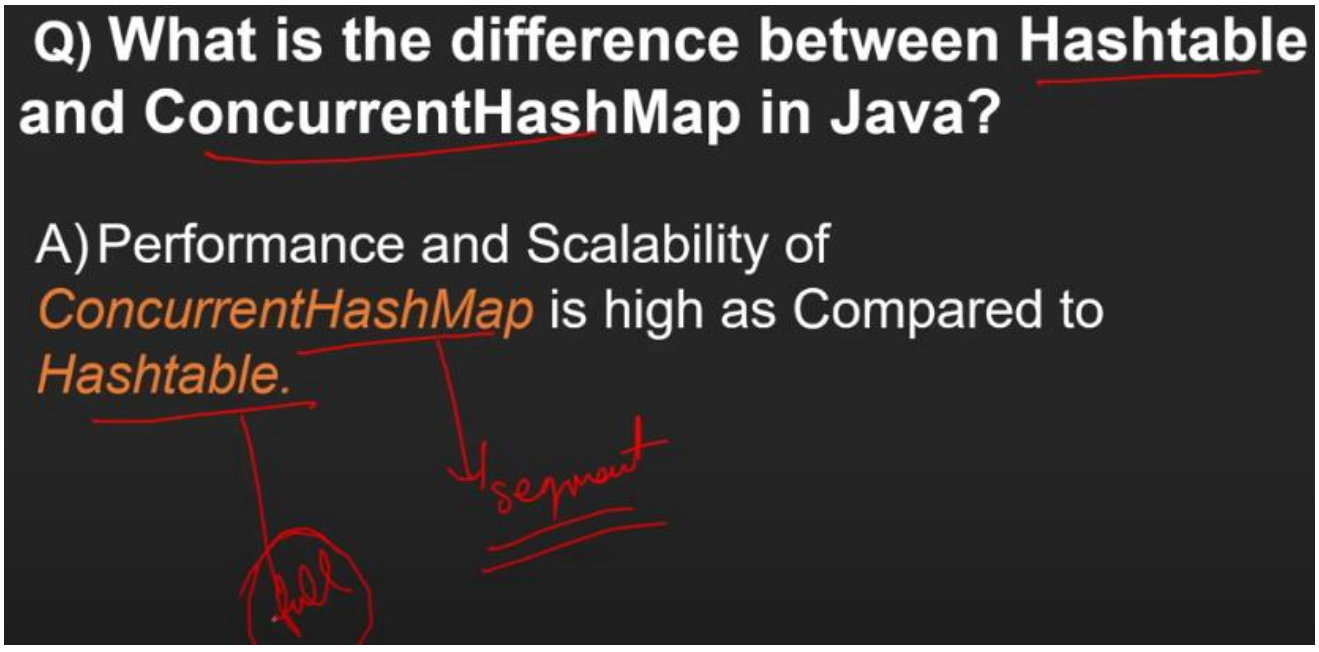




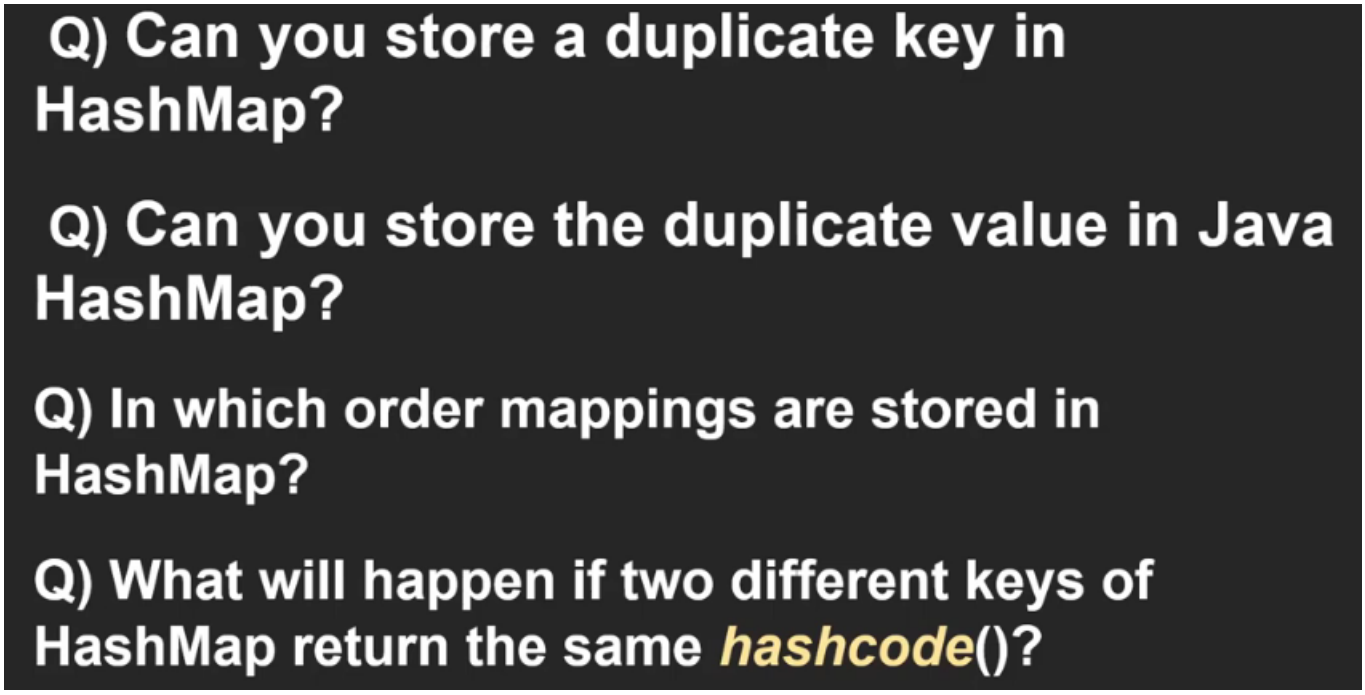


HashTable, Collections.synchrinizedMap(normal_map) and ConcurrentHashMap are both implementation on Map and are thread safe

```
try {
    // create a HashMap object
    Map<String, String> hMap
        = new HashMap<String, String>();

    // add elements into the Map
    hMap.put("1", "Welcome");
    hMap.put("2", "To");
    hMap.put("3", "Geeks");
    hMap.put("4", "For");
    hMap.put("5", "Geeks");

    System.out.println("Map : " + hMap);

    // Synchronizing the map
    Map<String, String> sMap
        = Collections.synchronizedMap(hMap);

    // printing the Collection
    System.out.println("Synchronized map is : "
                       + sMap);
}

catch (IllegalArgumentException e)
{
    System.out.println("Exception thrown : " + e);
}
```

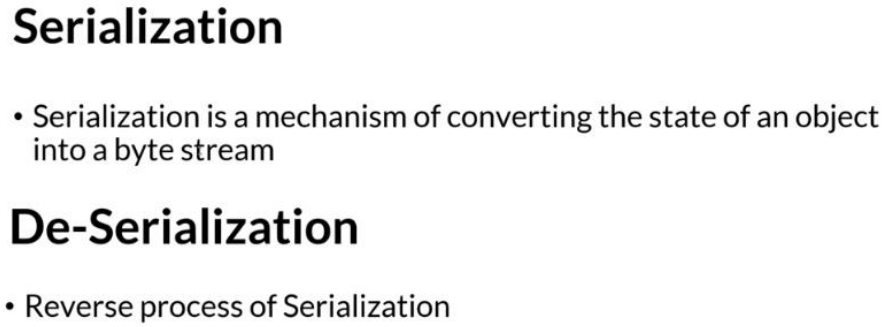


ConcurrentHashMap kuch segments ko lock karta where as Hashtable pure Object ko lock kar deta hai



1. No
2. Yes
3. No order
4. Collison (chaining used with ll)

Serialization and **Deserialization**: (Asked in IO)

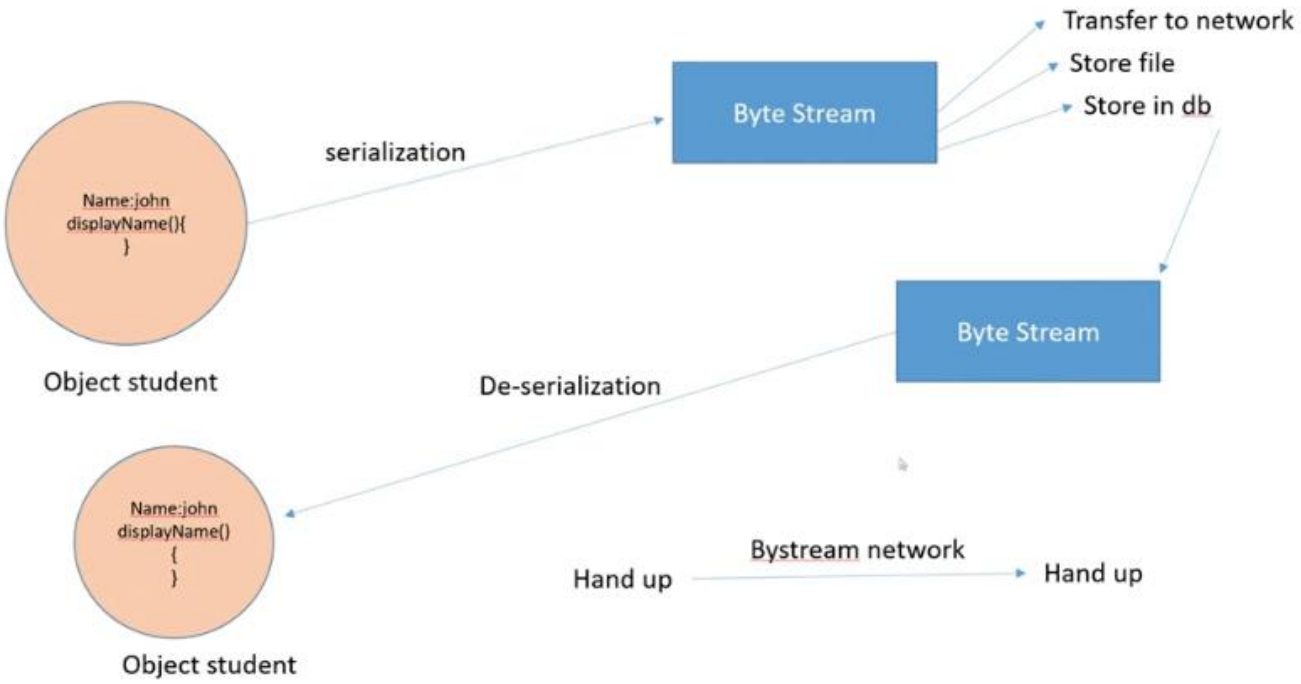


Object State meaning: Us object ke pass active properties hogi, attributes and behavior hogi

After doing Serialization we can transfer object state to **Network**, can store in **Database**, can save in **File**

Again by doing Deserialization we can access the state object received from network, fetched from database or retrieved from file.

When we have a object we cannot transfer it through network. First we have convert it to Byte stream by doing Serialization.



Same state ko transfer kar sakte hai and receiving system us same state ko access kar sakta hai

Serializable is a markup interface that tells if a class object is serializable or not.

```
public class Student implements Serializable {
```

Transient is a keyword which prevents a variable from serializable

Serialization

```java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

public class Serial {
    Run | Debug
    public static void main(String[] args) {

        Student st = new Student(name:"Arbaz", email:"arbaz@gmail.com",age:22, address:"Mumbai");

        try{
            FileOutputStream fos = new FileOutputStream(name:"ob.txt");

            ObjectOutputStream oos = new ObjectOutputStream(fos);

            oos.writeObject(st);

            fos.close();
            oos.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Deserialization

```java
public class Deserial {
    Run | Debug
    public static void main(String[] args) {
        try{
            FileInputStream fos = new FileInputStream(name:"ob.txt");

            ObjectInputStream ois = new ObjectInputStream(fos);

            Student st = (Student) ois.readObject();

            st.displayName();
            System.out.println(st.getName());
            System.out.println(st.getEmail());
            System.out.println(st.getAge());
            System.out.println(st.getAddress());
            fos.close();
            ois.close();
        }
        catch(ClassNotFoundException e){
            e.printStackTrace();
        }
        catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Important class for Serialization

FileOutputStream and ObjectOutputStream

Method: writeObject(obj)

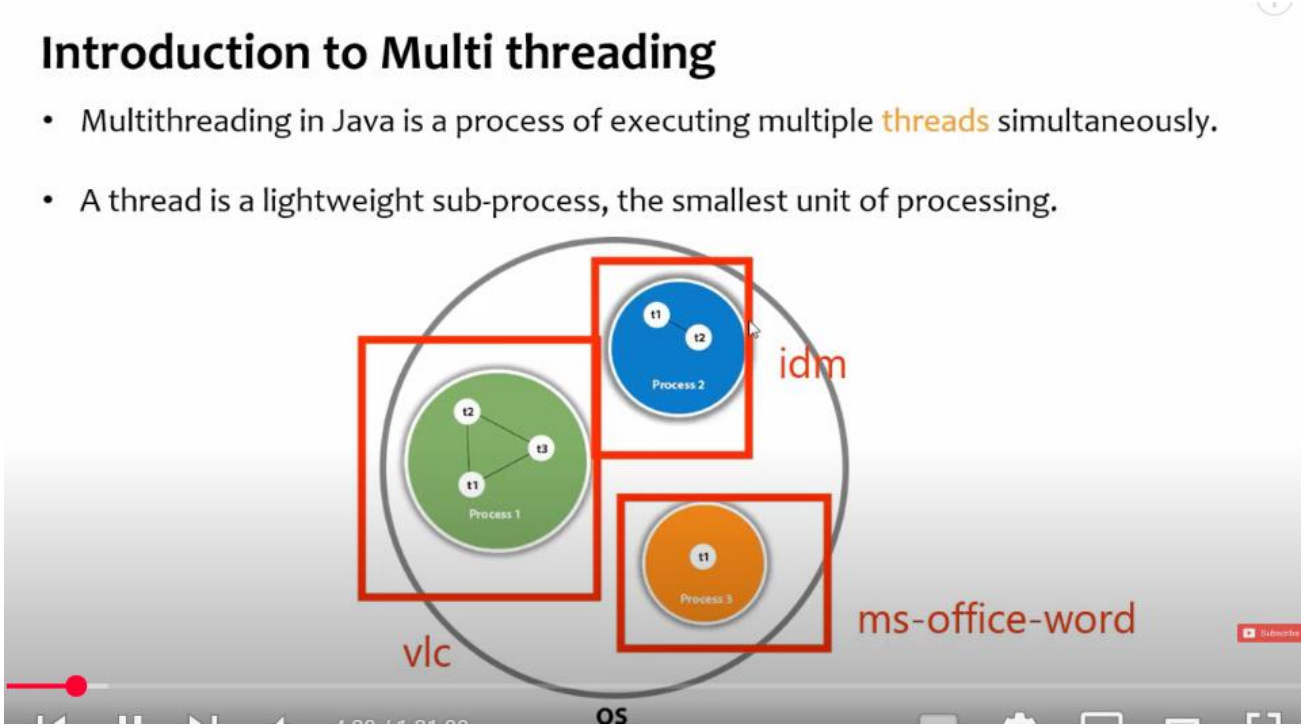
Important class for Deserialization

FileInputStream and ObjectInputStream

Method: readObject()

Multi -Threading (Java is Multi-Threading Programming Language**)**

Multi threading is used to achieve Multi tasking



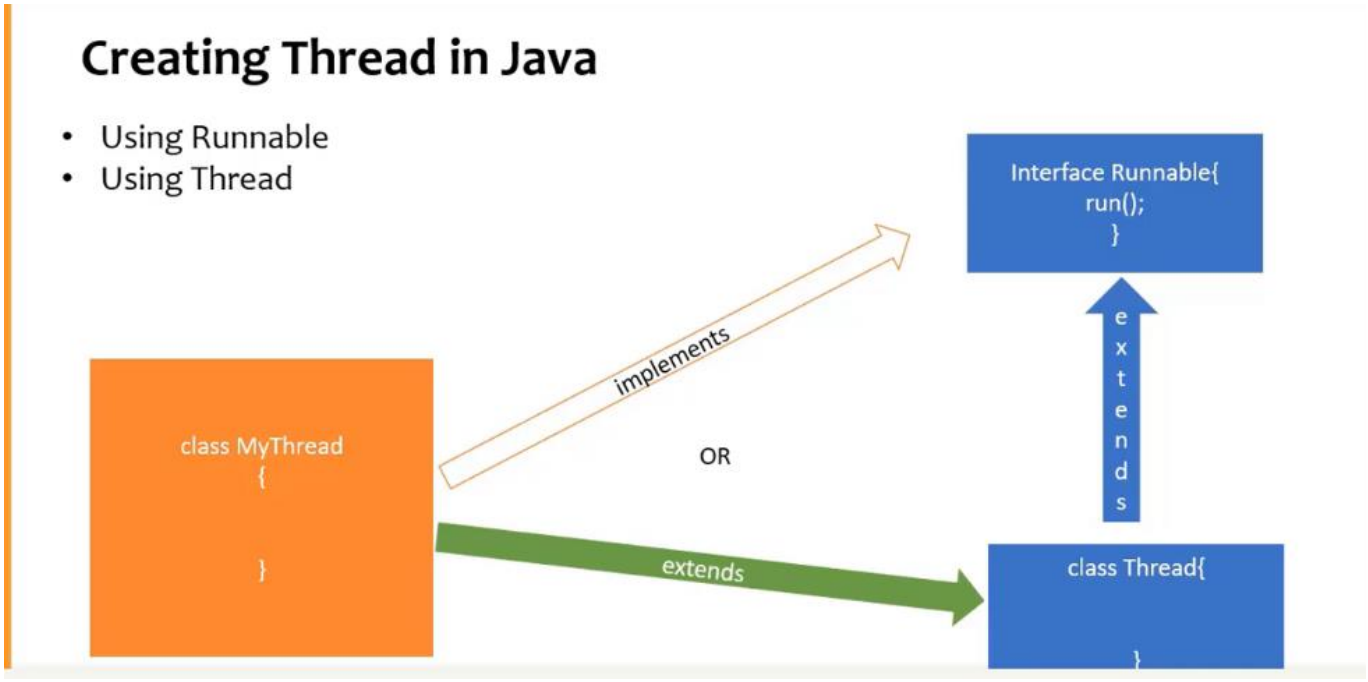
Multi threads forms a process.

IDM – Internet Download Manager

Multi Processing occurs at OS level when Multiple processes runs.

Multi Threading occurs within a single process. The application is developed using a language that supports multi threading. So Multi threading is programming level phenomenon occurs within a single process .

For a Process there needs to be at least one thread.



Both are in java.lang package so no need to explicitly import anything.

Starting Thread in Java

```
class MyThread implements Runnable
{
        public void rrun()
        {
    //task
        }

}
```

```
MyThread t=new MyThread();

Thread thread=new Thread(t);

thread.start()
```

```
class MyThread extends Thread
{
        public void rrun()
        {
    //task
        }

}
```

```
MyThread t=new MyThread();

t.start()
```

Best way is to use **Runnable** interface as we are implementing, we can also extend some other class. If we go with extending Thread class then no other class can be inherited.