Java String:

1) What is String in Java? Is String is data type?

String in Java is not a primitive data type like int, long or double. The string is a class or in more simple term a user defined type. This is confusing for someone who comes from C background. String is defined in java.lang package and wrappers its content in a character array. String provides equals() method to compare two String and provides various other methods to operate on String like `toUpperCase()` to convert String into upper case, `replace()` to replace String contents, `substring()` to get substring, `split()` to split long String into multiple String.

Q1. What is a String Pool ?

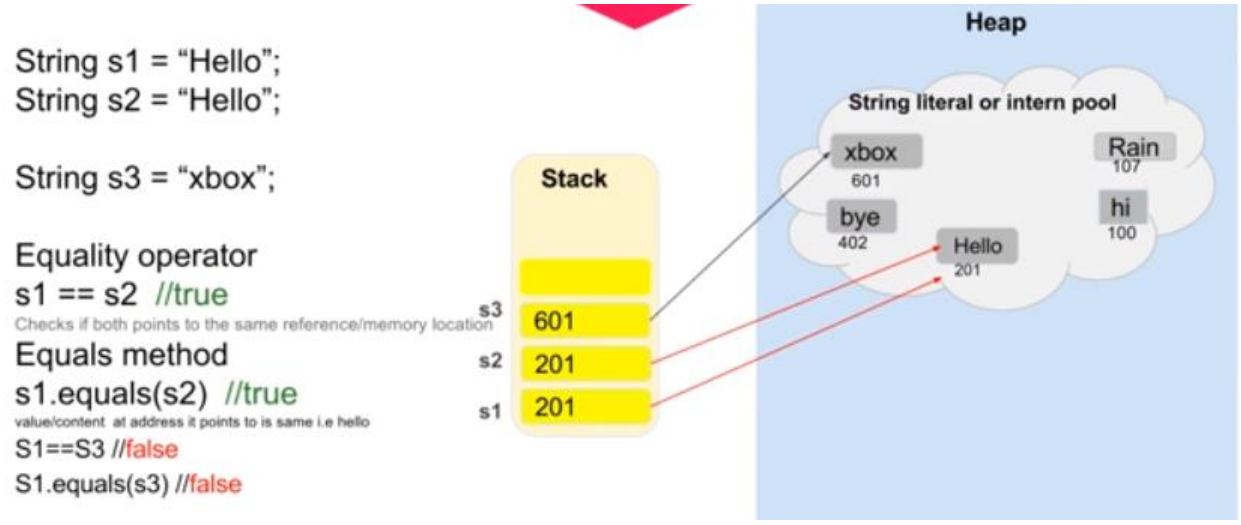
Ans. String pool (String intern pool) is a special storage area in Java heap. When a string literal is referred and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

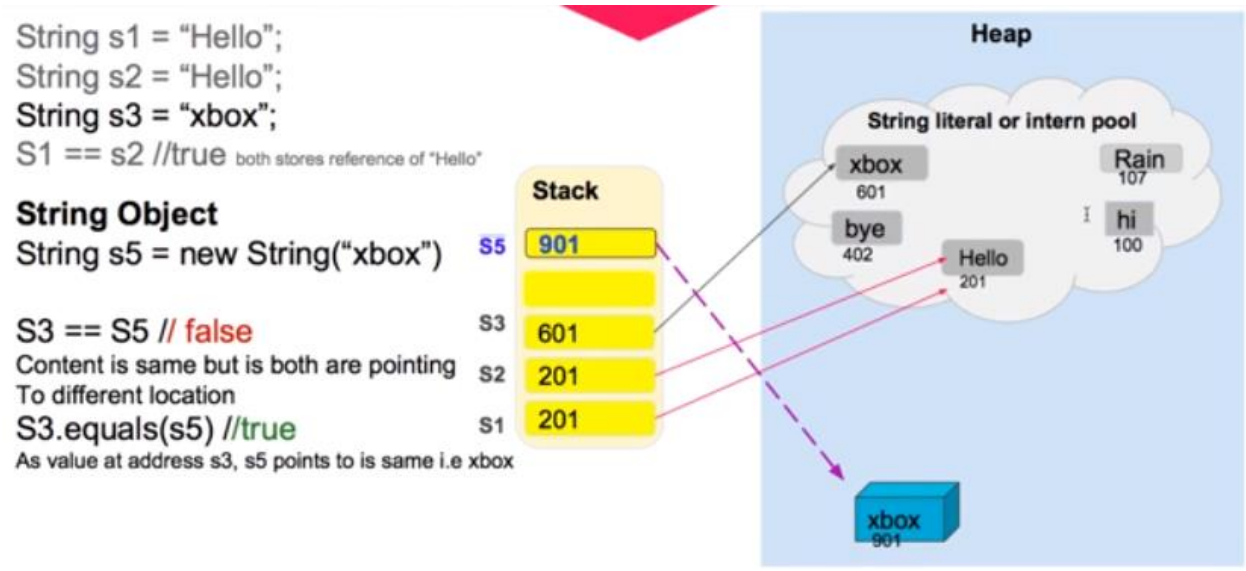
Q2. What are different ways to create String Object? Explain.

Ans. String str = new String("abc");
String str1 = "abc";

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.

When we use new operator, JVM creates the String object but don't store it into the String Pool. We can use intern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.





Q4. What will be the output of following Code ?

```
class BuggyBread {
   public static void main(String[] args)
   {
       String s2 = "I am unique!";
       String s5 = "I am unique!";

       System.out.println(s2 == s5);
   }
}
```

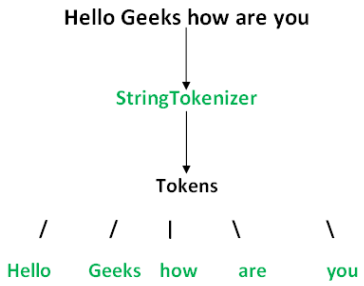
Ans. **Output will be true**, due to String Pool, both will point to a same String object.

Q5. What will be the output of following Code ?

```
class BuggyBread {
   public static void main(String[] args)
   {
       String s2 = new String("I am unique!");
       String s5 = new String("I am unique!");

       System.out.println(s2 == s5);
   }
}
```

Ans. **Output will be false**, as they are two different objects. String Pool won't be referred here.

StringTokenizer: The java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string.



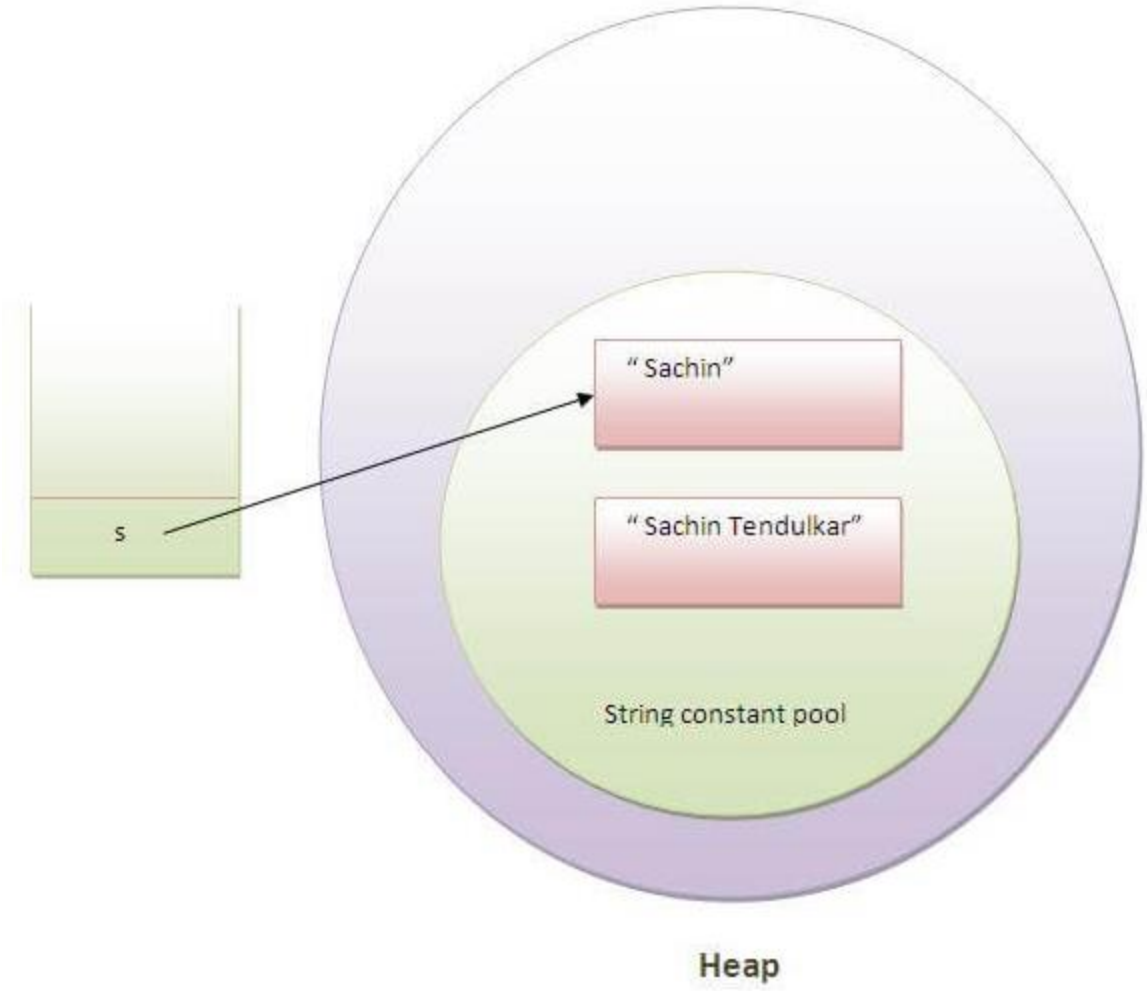
Why is String immutable in Java?

1. String pool is possible only because String is immutable in java, this way Java Runtime saves a lot of java heap space because different String variables can refer to same String variable in the pool. If String would not have been immutable, then String interning would not have been possible because if any variable would have changed the value, it would have been reflected to other variables also.
2. If String is not immutable then it would cause severe security threat to the application. For example, database username, password are passed as String to get database connection and in socket programming host and port details passed as String. Since String is immutable it's value can't be changed otherwise any hacker could change the referenced value to cause security issues in the application.
3. Since String is immutable, it is safe for multithreading and a single String instance can be shared across different threads. This avoid the usage of synchronization for thread safety, Strings are implicitly thread safe.
4. Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader. For example, think of an instance where you are trying to load java.sql.Connection class but the referenced value is changed to myhacked.Connection class that can do unwanted things to your database.
5. Since String is immutable, its **hashcode** is cached at the time of creation and it doesn't need to be calculated again. This makes it a great candidate for key in a Map and it's processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

In your example, if `String` was mutable, then consider the following example:

```
String a="stack";
System.out.println(a);//prints stack
a.setValue("overflow");
System.out.println(a);//if mutable it would print overflow
```

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```



1) String Pool

Java designer knows that `String` is going to be most used data type in all kind of Java applications and that's why they wanted to optimize from start. One of key step on that direction was idea of storing String literals in String pool. Goal was to reduce temporary `String` object by sharing them and in order to share, they must have to be from Immutable class. You can not share a mutable object with tw parties which are unknown to each other. Let's take an hypothetical example, where two reference variable is pointing to same `String` object:

```
String s1 = "Java";
String s2 = "Java";
```

Now if `s1` changes the object from `"Java"` to `"C++"`, reference variable also got value `s2="C++"`, which it doesn't even know about it. By making String immutable, this sharing of String literal was possible. In short, key idea of String pool can not be implemented without making String final or Immutable in Java.

Pros and Cons of String being Immutable or Final in Java

Apart from above benefits, there is one more advantage that you can count due to String being final in Java. It's one of the most popular object to be used as key in hash based collections e.g. HashMap and Hashtable. Though immutability is not an absolute requirement for `HashMap` keys, its much more safe to use Immutable object as key than mutable ones, because if state of mutable object is changed during its stay inside HashMap, it would be impossible to retrieve it back, given it's `equals()` and `hashCode()` method depends upon the changed attribute. If a class is Immutable, there is no risk of changing its state, when it is stored inside hash based collections. Another significant benefits, which I have already highlighted is its thread-safety. Since String is immutable, you can safely share it between threads without worrying about external synchronization. It makes concurrent code more readable and less error prone.

Custom Immutable Class:

Immutable Class Banane ke Rules

1. **Class ko `final` declare karein:** Taaki koi dusri class ise extend na kar sake aur methods ko override na kar sake.
2. **Saare fields ko `private` aur `final` rakhein:** * `private` taaki direct access na ho.
   - `final` taaki initialization ke baad value change na ho sake.
3. **Setter methods na banayein:** Object ki state change karne ka koi tarika nahi hona chahiye.
4. **Parameterized Constructor use karein:** Saare fields ko constructor ke zariye hi initialize karein.
5. **Deep Copy (Deep Cloning):** Agar class mein koi **Mutable Object** (jaise `Date`, `List`, `Map`) hai, toh constructor aur getter mein uski "deep copy" return karein.

```java
public final class ImmutableStudent {

    private final int id;
    private final String name;

    public ImmutableStudent(int id, String name) {
        this.name = name;
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

```java
public final class ImmutableStudent {

    // 2. Fields are private and final
    private final String name;
    private final int regNo;
    private final Date joiningDate;

    // 3. Constructor initializing fields
    public ImmutableStudent(String name, int regNo, Date date) {
        this.name = name;
        this.regNo = regNo;
        // 5. Deep Copy for mutable objects
        this.joiningDate = new Date(date.getTime());
    }

    public String getName() {
        return name;
    }

    public int getRegNo() {
        return regNo;
    }

    // 5. Getter returning a clone, not the original reference
    public Date getJoiningDate() {
        return new Date(joiningDate.getTime());
    }
```

StringBuffer:

In Java, **StringBuffer** is a peer class of String that provides much of the same functionality, but with one critical difference: **it is mutable**.

While a standard String is fixed in size and content once created, a StringBuffer can be modified—you can append, insert, or delete characters without creating a new object every time.

- **Mutable Content:** You can modify the string (append, delete, or insert) without creating a new object, which saves memory during heavy string manipulations.
- **Thread Safety:** All major methods are **synchronized**, making it safe to use when multiple threads are accessing or modifying the same string object simultaneously.
- **Heap Allocation:** Unlike `String` literals that live in the String Constant Pool, `StringBuffer` objects are always allocated in the **Heap memory**.
- **Initial Capacity:** It starts with a default capacity of **16 characters** (plus the initial string length) and automatically expands its internal buffer as needed.
- **Inheritance:** It extends the `Object` class and implements the `CharSequence`, `Appendable`, and `Serializable` interfaces, ensuring it integrates well with standard Java utility methods.

1. **once we creates a string object we can't perform any changes in the existing object. if we are trying to perform any changes with those changes a new object will be created. this non changeable nature is nothing but immutability of the string object.**

2. **Once we creates a StringBuffer object we can perform any type of changes in the existing object. this changeble is nothing but mutability of the StringBuffer object.**

StringBuilder:

In Java, **StringBuilder** is a helper class used to create and manipulate mutable (changeable) sequences of characters.

If you use a standard String, every time you "change" it (like adding a word), Java actually creates a brand-new object in memory because Strings are **immutable**. For loops or heavy text processing, this is very slow and wastes memory. StringBuilder solves this by modifying the same object in place.

- **Mutable:** You can add, remove, or replace characters without creating new objects.
- **Not Thread-Safe:** It is designed for use by a single thread. This makes it faster than its older sibling, `StringBuffer`.
- **Memory Efficient:** It uses a dynamic array (buffer) that grows as needed.

StringBuffer	StringBuilder
1. Every method present in StringBuffer is synchronized.	1.No method present in StringBuilder is synchronized .
2. At a time only one thread is allow to operate on StringBuffer object. Hence StringBuffer object is Thread safe	2. At a time multiple threads are allow to operate on StringBuilder object and hence StringBuilder object is not Thread Safe.
3. It Increases waiting time of threads and hence relatively performance is low.	3. Threads are not required to wait to operate on StringBuilder object and hence relatively performance is high.
4. Introduced in 1.0 version	4. Introduced in 1.5 version

Object Class:

In Java, the **Object** class is the "mother of all classes." It sits at the very top of the class hierarchy in the java.lang package.

Every single class you create—or that is built into Java—implicitly inherits from Object. If you don't use the extends keyword, Java automatically adds extends Object for you.

Why is it so important?

Because every class is a child of `Object`, all Java objects share a common set of behaviors. This allows for **Polymorphism**: you can create a list like `ArrayList<Object>` that can hold *anything*—Strings, Integers, or custom user objects—because they are all technically "Objects."

Most Frequently Used Methods

Since these are inherited by every class, you will often **override** them to change how your specific class behaves.

Method	Purpose	Default Behavior
`toString()`	Returns a string version of the object.	Returns `ClassName@HashCode` (usually not very helpful).
`equals(Object obj)`	Checks if two objects are "equal."	Checks if they are the exact same memory address (`==`).
`hashCode()`	Returns a unique integer ID for the object.	Based on the memory address (used in HashMaps).
`getClass()`	Returns the "Class" object (metadata).	Tells you the actual runtime name of the class.
`clone()`	Creates a copy of the object.	Throws an error unless you implement the `Cloneable` interface.

Threading Methods

The `Object` class also contains methods used for synchronization and communication between threads:

- `wait()` : Tells the current thread to give up its lock and wait.
- `notify()` : Wakes up a single thread waiting on this object.
- `notifyAll()` : Wakes up all threads waiting on this object.

Singleton class:

Java mein ek aisi class hoti hai jiska poore program mein sirf **ek hi object (instance)** ban sakta hai.

Agar aap dusri baar object banane ki koshish karenge, toh wahi pehla wala object hi wapas mil jayega. Naya object nahi banega.

The Singleton's purpose is to control object creation, limiting the number of objects to only one. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources, such as database connections or sockets.

```
// File Name: Singleton.java
public class Singleton {

   private static Singleton singleton = new Singleton( );

   /* A private Constructor prevents any other
    * class from instantiating.
    */
   private Singleton() { }

   /* Static 'instance' method */
   public static Singleton getInstance( ) {
      return singleton;
   }

   /* Other methods protected by singleton-ness */
   protected static void demoMethod( ) {
      System.out.println("demoMethod for singleton");
   }
}
```

Here is the main program file where we will create a singleton object –

```
// File Name: SingletonDemo.java
public class SingletonDemo {

   public static void main(String[] args) {
      Singleton tmp = Singleton.getInstance( );
      tmp.demoMethod( );
   }
}
```

This will produce the following result –

Output

```
demoMethod for singleton
```

Example 2

Following implementation shows a classic Singleton design pattern –

```
public class ClassicSingleton {

   private static ClassicSingleton instance = null;
   private ClassicSingleton() {
      // Exists only to defeat instantiation.
   }

   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
}
```

The ClassicSingleton class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

Here, ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.

Question 5: Can you write critical section code for a singleton? (answer)

This core Java question is a follow-up of the previous question and expects the candidate to write a Java singleton using the Double-Checked Locking Pattern.

Remember to use a volatile variable to make Singleton thread-safe.

Here is the code for a critical section of a thread-safe singleton class using the Double-Checked Locking idiom:

```
public class Singleton {
    private static volatile Singleton _instance;
    /**
     * Double checked locking code on Singleton
     * @return Singelton instance
     */
    public static Singleton getInstance() {
        if (_instance == null) {
            synchronized(Singleton.class) {
                if (_instance == null) {
                    _instance = new Singleton();
                }
            }
        }
        return _instance;
    }
}
```

Interface:

Interface is a reference type, similar to a class, that is used to specify a set of behaviors that a class must implement. Think of it as a **contract** or a **blueprint** for a class.

It tells a class *what* to do, but not *how* to do it.

Core Characteristics

- **Abstraction:** Interfaces provide total abstraction (before Java 8). They cannot be instantiated—meaning you can't create an object of an interface.
- **Methods:** By default, methods in an interface are `public` and `abstract` (they don't have a body).
- **Variables:** Any variables declared in an interface are implicitly `public`, `static`, and `final` (constants).
- **Multiple Inheritance:** A Java class can only inherit from one superclass, but it can **implement multiple interfaces**. This is how Java supports multiple inheritance.

Why Use Interfaces?

Feature	Benefit
Security	Hide implementation details and only expose the "signature" of the methods.
Loose Coupling	Programs become more flexible because you can swap different implementations of the same interface without changing the calling code.
Standardization	Ensures that different classes follow the same structure (e.g., all `Printable` objects must have a `print()` method).

Evolution of Interfaces (Java 8+)

Modern Java has introduced more flexibility to interfaces:

- **Default Methods:** Allow you to add new functionality to interfaces without breaking existing classes that implement them.
- **Static Methods:** Allow you to define helper methods that belong to the interface class rather than the object instance.

Marker Interface:

Marker Interface ek aisa interface hota hai jo bilkul **khali (empty)** hota hai. Isme koi method ya variable nahi hota.

Isko aap ek **"Label"** ya **"Stamp"** ki tarah samajhiye.

Ek Real-Life Example (Courier Package)

Sochiye aap ek courier bhej rahe hain.

1. Agar aap box par **"FRAGILE"** (kaanch ka saaman) ka sticker laga dete hain, toh courier wala usse dhyan se uthayega.
2. Sticker ke andar kuch likha nahi hota, wo bas ek **ishara (signal)** hai.

Java mein Marker Interface wahi "Sticker" hai. Jab ek class isse implement karti hai, toh wo JVM (Java Virtual Machine) ko ek signal deti hai ki: *"Mere saath kuch special treatment karo."*

Java Example: Serializable

Sabse zyada use hone wala marker interface `Serializable` hai.

Java

```java
import java.io.Serializable;

// Humne 'Serializable' ka sticker laga diya
class Student implements Serializable {
    int id;
    String name;
}
```

Yaha kya hua?

- `Serializable` interface ke andar koi method nahi hai.
- Lekin jaise hi humne `implements Serializable` likha, Java samajh gaya ki is class ke objects ko hum **File mein save** kar sakte hain ya **Network par bhej** sakte hain.
- Agar aap ye "sticker" nahi lagate, toh Java error de dega.

Main Points (Asaan Points mein)

- **Khali Interface:** Iske andar `public void methods()` jaisa kuch nahi hota.
- **Pechaan (Identity):** Ye class ko ek special identity deta hai.
- **Runtime Check:** Java check karta hai: `if (obj instanceof Serializable)`, agar haan, toh wo apna kaam shuru karta hai.

Interface & Abstract Class

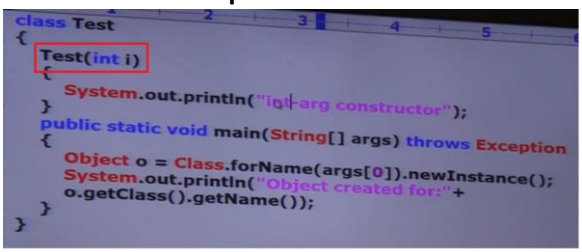
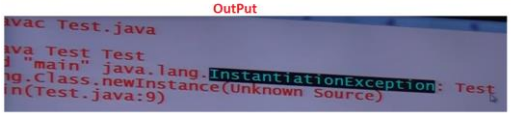
Interface	Abstract Class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
2. Inside Interface every method is always public and abstract whether we are declaring or not. Hence interface is also considered as 100% pure Abstract class.	2. Every method present in Abstract class need not be public and Abstract. In addition to abstract methods we can take concrete methods also.
3. We can't declare interface method with the following modifiers. Public ---> private, protected, Abstract ---> final, static, synchronized, native, strictfp	3. There are no restrictions on Abstract class method modifiers.
4. Every variable present inside interface is always public, static and final whether we are declaring or not.	4. The variables present inside Abstract class need not be public static and final.
5. We can't declare interface variables with the following modifiers. private, protected, transient, volatile.	5. There are no restrictions on Abstract class variable modifiers.
6. For Interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6. For Abstract class variables it is not required to perform initialization at the time of declaration.
7. Inside interface we can't declare instance and static blocks. Otherwise we will get compile time error.	7. Inside Abstract class we can declare instance and static blocks.
8. Inside interface we can't declare constructors.	8. Inside Abstract class we can declare constructor, which will be executed at the time of child object creation.

Overloading & Overriding:

Property	Overloading	Overriding
1. Method names	Must be same	Must be same
2. Argument Types	Must be different (at least order)	Must be same (Including order)
3. Method signatures	Must be different	Must be same.
4. Return Type	No Restrictions	Must be same but this rule is applicable until 1.4version only. From 1.5v onwards co-variant return types are allowed.
5.private,static & final methods	Can be overloaded	Cannot be overridden
6. Access modifiers	No Restrictions	We can't reduce scope of Access modifier but we can increase.
7. throws clause	No Restrictions	If child class method throws any checked exception compulsory parent class method should throw the same checked exception are its parent otherwise we will get compile time error but there are no restrictions for Unchecked Exceptions.
8. Method Resolution	Always takes care by compiler based on reference type.	Always takes care by JVM based on Runtime object.
9.Also Known as	Compile time polymorphism or static polymorphism or early binding.	Runtime polymorphism, dynamic polymorphism or late binding.

new operator and newInstance() Method:

Difference	new operator	newInstance()
Object Creation	Test t=new Test()	Objtect obj=Class.forName("className").newInstance()

Argument Constructor()	Test t=new Test();
	Test t=new Test(10,10);
	If you use no-argument constructor compiler call no-argument constructor
		.newInstance()
		{
		no-argument constructor() only
		}
		We can't pass parameterized contractor
		If we pass parameter then we will get "InstantiationException"
		 
		
		OutPut
		 

Exception Difference : If class is **not** there	Test t=new Test(); If Test class is not there then we will get "**NoClassDefFountExcption**" Note: NoClassDefFoundException is the UncheckedException.	Objtect obj=Class.forName("Test").newInstance(); If Test class is not there then we will get "**ClassNotFoundException**" Note: ClassNotFoundExption is the Checked Exception.

"==" & ".equals()" method.

Difference	Double equals operator(==)	equals() method
Definition	Double equals operator(==) is for "**Reference Comparison**"	Equals method is for "**content comparison**"

Example	String s1=new String(“Rahul”); String s2=new String(“Rahul”); **System.out.println(s1==s2) ->False**	String s1=new String(“Rahul”); String s2=new String(“Rahul”); **System.out.println(s1.equals(s2))->True**

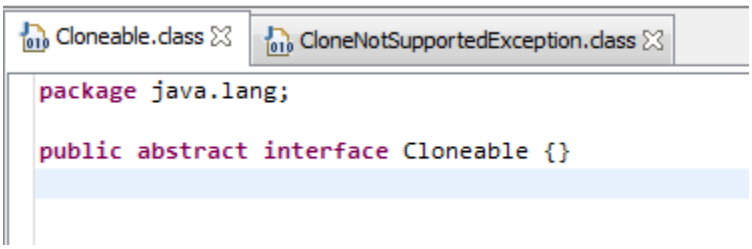
Object Cloning:

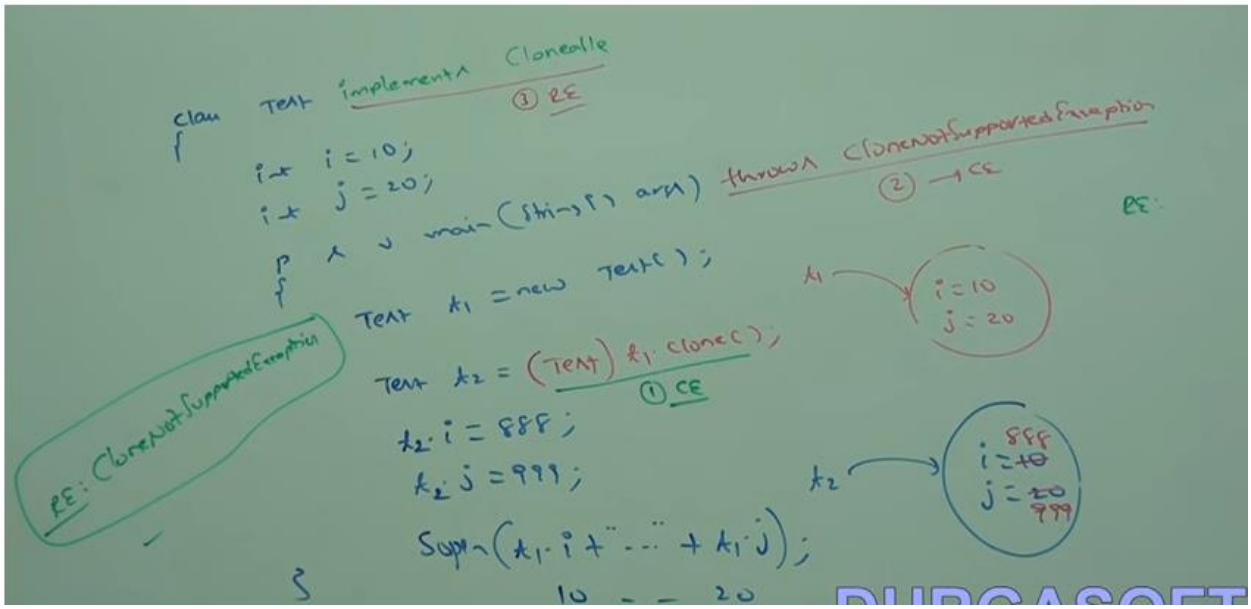
The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

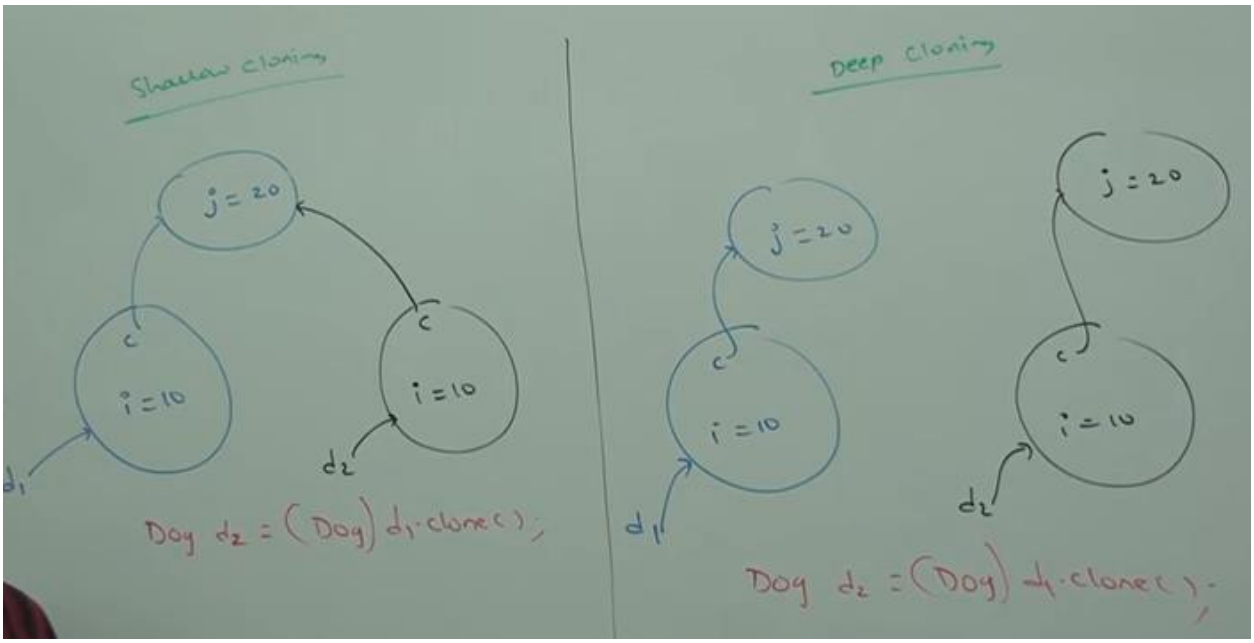
The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

```
protected Object clone() throws CloneNotSupportedException
```







Note : Shallow cloning uses Object class clone() method.

Dog.java

```java
/* Shallow Cloning - cloning only primitive type variable not the instance type variable  */
package com.example;

class Cat {
    public int j;

    public Cat(int j) {
        this.j = j;
    }
}

public class Dog implements Cloneable {
    public Cat c;
    public int i;

    public Dog(Cat c, int i) {
        this.c = c;
        this.i = i;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat(10);
        Dog d1 = new Dog(c, 5);
        Dog d2 = (Dog) d1.clone();

        System.out
                .println("Before instance variable(cat) value change--->d1.c.j="
                        + d1.c.j + "----d2.c.j=" + d2.c.j);
        /* after change cat j value */
        d2.c.j = 11;
        System.out
                .println("After instance variable(cat) value change--->d1.c.j="
                        + d1.c.j + "----d2.c.j=" + d2.c.j);
        /* called as shallow cloning */
    }
}
```

Console

<terminated> Dog [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (May 1, 2018, 1:58:02 PM)

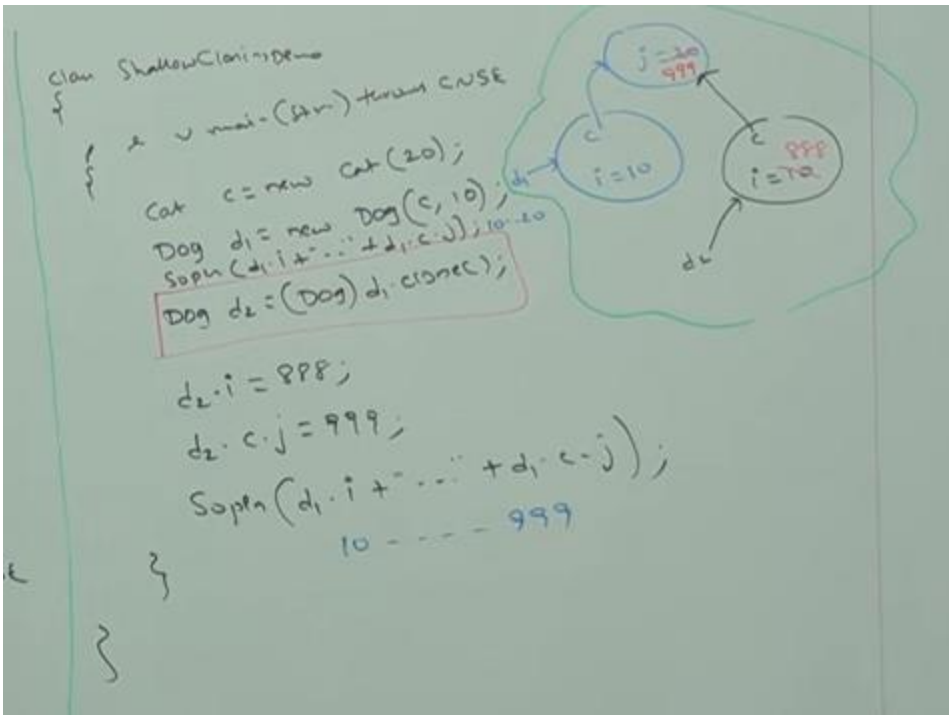
```
Before instance variable(cat) value change--->d1.c.j=10----d2.c.j=10
After instance variable(cat) value change--->d1.c.j=11----d2.c.j=11
```

```
class Cat
{
    int j;
    Cat(int j)
    {
        this.j = j;
    }
}
class Dog implements Cloneable
{
    Cat c;
    int i;
    Dog(Cat c, int i)
    {
        this.c = c;
        this.i = i;
    }
```

```
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }

}
```

```
class ShallowCloning
{
    public static void main(String[] args) throws CloneNots
    {
        Cat c = new Cat(20);
        Dog d1 = new Dog(c,10);
        System.out.println(d1.i+"....."+d1.c.j);
        Dog d2 = (Dog)d1.clone();
        d2.i = 888;
        d2.c.j = 999;
        System.out.println(d1.i+"....."+d1.c.j);
    }
}
```

```
C:\durga_classes>javac ShallowCloning.java

C:\durga_classes>java ShallowCloning
10.....20
10.....999

C:\durga_classes>_
```



In Shallow cloning – By using cloned object reference if we perform any change to the content object then those changes will be reflected to the main object.

To overcome this problem we should go for "Deep Cloning".

Deep Cloning: The process of creating exactly duplicate independent copy including "contend object" is called deep cloning.

In deep cloning if the main object contend any primitive variables then in the cloned object duplicate copy will be created.

If the main object contend any reference variable then corresponding contend object will be created in the cloned copy.

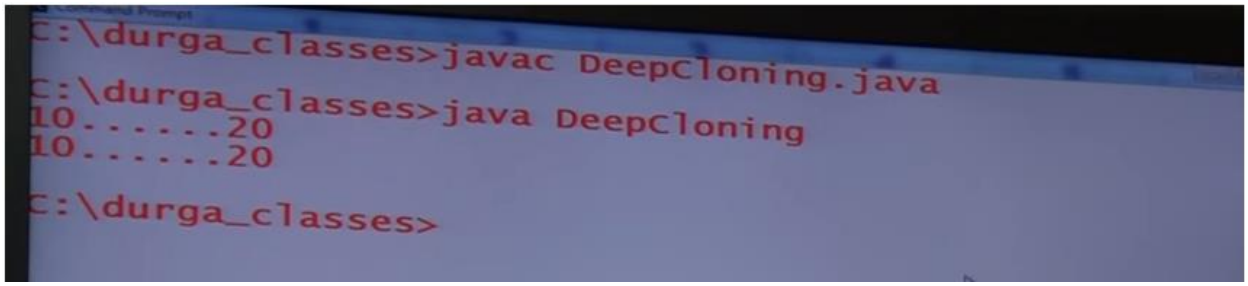
By default object class clone() method mint for shallow cloning, but we can implement deep cloning explicitly by overriding clone() method in over class.

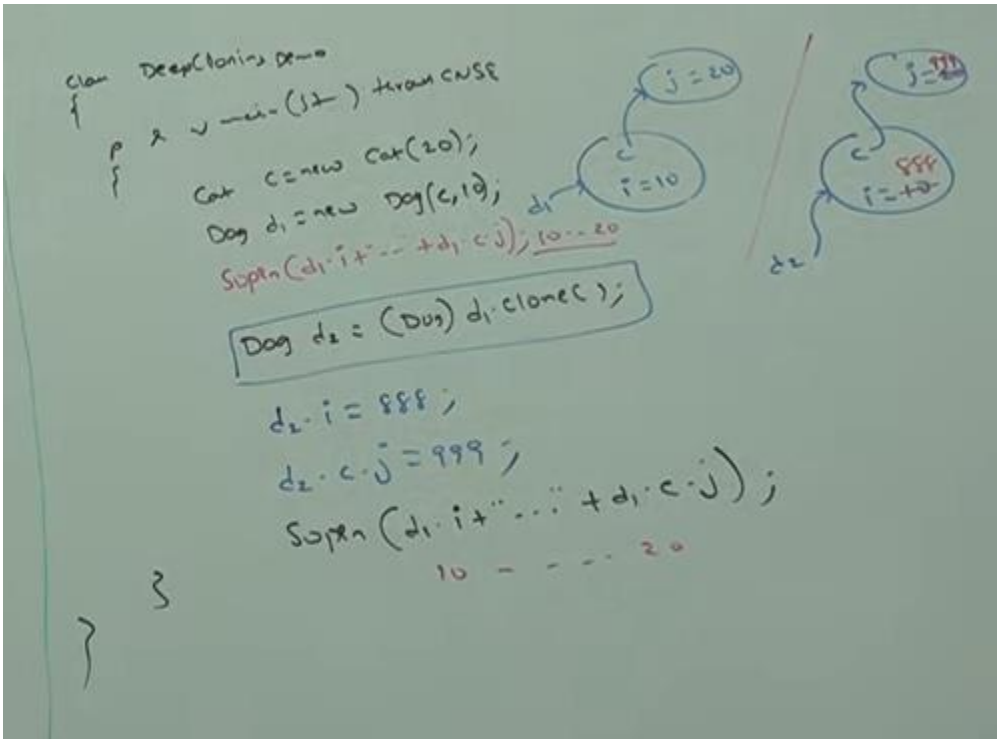
```
class Cat
{
   int j;
   Cat(int j)
   {
      this.j = j;
   }
}
class Dog implements Cloneable
{
   Cat c;
   int i;

   Dog(Cat c, int i)
   {
      this.c = c;
      this.i = i;
   }
```



```
   public Object clone() throws CloneNotSupportedException
   {
      Cat c1 = new Cat(c.j);
      Dog d = new Dog(c1,i);
      return d;
   }
}
```

```
class DeepCloning
{
   public static void main(String[] args) throws CloneNotSupp
   {
      Cat c = new Cat(20);
      Dog d1 = new Dog(c,10);
      System.out.println(d1.i+"....."+d1.c.j);
      Dog d2 = (Dog)d1.clone();
      d2.i = 888;
      d2.c.j = 999;
      System.out.println(d1.i+"....."+d1.c.j);

   }
}
```





Generics:

Generics Java ka ek bahut powerful feature hai jo aapko **Type-safe** code likhne mein madad karta hai.

Asaan shabdon mein, Generics ka matlab hai ki aap ek hi class, interface, ya method ko alag-alag data types (Integer, String, User-defined) ke saath use kar sakte hain, bina baar-baar naya code likhe.

Type Parameters tab use hote hain jab hum koi cheez **define** kar rahe hote hain, aur **Wildcards** tab use hote hain jab hum kisi cheez ko **use** kar rahe hote hain.

1. Type Parameters (`<T>`)

Type Parameters ek tarah ke **placeholders** hote hain. Jab aap ek class ya method likhte hain aur aap chahte hain ki wo kisi bhi data type (Integer, String, etc.) ke saath kaam kare, toh aap Type Parameter ka use karte hain.

- **Syntax:** `<T>`, `<E>`, `<K, V>`
- **Kab use karein:** Jab aapko class, interface, ya method ka "template" banana ho.

2. Wildcards (`<?>`)

Wildcard ka matlab hota hai **"unknown type"**. Ye tab use hota hai jab aapko ye nahi pata (ya parwah nahi hoti) ki andar kaun sa exact type hai, ya phir jab aapko inheritance (polymorphism) handle karni ho.

- **Syntax:** `?`
- **Teen tarah ke hote hain:**
  1. **Unbounded Wildcard (`<?>`):** Kisi bhi type ke liye.
  2. **Upper Bounded (`<? extends Number>`):** Sirf Number ya uske subclasses (Integer, Double) ke liye.
  3. **Lower Bounded (`<? super Integer>`):** Sirf Integer ya uske parent classes (Number, Object) ke liye.

Generics in Java enable you to specify type parameters when defining classes, interfaces and methods. Here type parameter is the type of data (class or interface) which these classes, interfaces and methods will operate upon.

As example, when you say **List<T>** that means a list which will store elements of type T. Since you have specified a type parameter so this List interface is generic and T is the type parameter which specifies the type of element stored in the **List**.

Generic class format
A generic class is defined with the following format:

```
class name<T1, T2, ..., Tn> {
    /* ... */
}
```

Type-safe ka matlab hota hai ki compiler code run hone se pehle hi check kar leta hai ki aap sahi type ka data use kar rahe hain ya nahi.

Asaan shabdon mein: **"Jis type ka dabba hai, usme wahi saaman jayega."**

Agar aapne ek dabba `String` ke liye banaya hai, toh Java aapko usme `Integer` daalne hi nahi dega. Agar aap galti karenge, toh program chalne se pehle hi (Compile-time par) **Error** dikha dega.

1. Generics Kyun Chahiye? (The Problem)

Pehle (Java 5 se pehle), `ArrayList` mein hum kuch bhi daal sakte the kyunki wo `Object` type leta tha. Isse do badi problems hoti thi:

1. **Type Casting:** Har baar data nikaalte waqt use convert karna padta tha.
2. **Runtime Errors:** Agar galti se `Integer` ki list mein `String` daal di, toh program run hote waqt crash ho jata tha (`ClassCastException`).

Generics ne in dono problems ko khatam kar diya.

4. Bounded Type Parameters (Restrictions)

Kabhi-kabhi hum chahte hain ki hamara Generic har kisi ke liye kaam na kare, sirf kuch specific classes ke liye kare.

- **Example:** Agar aap ek `Calculator` class bana rahe hain, toh aap chahenge ki `T` sirf **Numbers** (`Integer` , `Double`) ho, `String` nahi.
- **Syntax:** `<T extends Number>`

Situation 1: Bina Generics ke (Mushkil Raasta)

Agar aapko `Integer` ke liye alag aur `String` ke liye alag dabba chahiye, toh aapko do alag classes banani padengi:

Java

```
// Integer ke liye alag class
class IntegerContainer {
    Integer data;
    void add(Integer d) { this.data = d; }
}

// String ke liye alag class
class StringContainer {
    String data;
    void add(String d) { this.data = d; }
}
```

Nuksan: Agar kal ko `Double` ya `Student` class ke liye container chahiye, toh phir se naya code likho. Isse "Code Duplication" hoti hai.

Situation 2: Generics ke saath (Asaan Raasta)

Hum ek hi class banayenge aur usme ek placeholder `<T>` daal denge. `T` ka matlab hai **"Type"**—jo baad mein decide hoga.

Java

```
// Ek hi generic class sabke liye
class MyContainer<T> {
    T data;

    void add(T d) {
        this.data = d;
    }

    T get() {
        return data;
    }
}
```

Iska Use kaise karein?

Ab aap is ek hi class se alag-alag type ke dabbe bana sakte hain:

Java

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

        // 1. Integer wala dabba banaya
        MyContainer<Integer> intDabba = new MyContainer<>();
        intDabba.add(100);

        // 2. String wala dabba banaya
        MyContainer<String> stringDabba = new MyContainer<>();
        stringDabba.add("Gemini");

        // 3. Float wala dabba banaya
        MyContainer<Float> floatDabba = new MyContainer<>();
        floatDabba.add(10.5f);

        System.out.println(intDabba.get());    // Output: 100
        System.out.println(stringDabba.get()); // Output: Gemini
    }
```

Summary: Asal fayda kya hua?

1. **Code Reusability:** Humne `MyContainer` class sirf **ek baar** likhi, lekin use **hazaron** tarah ke data types ke saath kar liya.
2. **Type Safety:** Jab humne `MyContainer<Integer>` likha, toh Java ne fix kar diya ki is dabba mein sirf numbers hi jayenge. Agar aap `intDabba.add("Hello")` likhenge, toh compiler turant error de dega.

Wildcards in Generics (?):

Java mein **Wildcard (?)** ka matlab hota hai **"Anjaan Type" (Unknown Type)**.

Jab aap code likhte hain aur aapko ye nahi pata (ya farq nahi padta) ki samne wala kaunsa data bhejega, tab aap ? ka use karte hain.

Jab humein pata nahi hota ki kaunsa type aayega, hum `?` ka use karte hain.

- `<?>` **(Unbounded):** Kuch bhi aa sakta hai.
- `<? extends Number>` **(Upper Bound):** Sirf `Number` ya uske bachche (subclasses).
- `<? super Integer>` **(Lower Bound):** Sirf `Integer` ya uske parent classes.

PECS(**P**roducer **E**xtends, **C**onsumer **S**uper.) Rule

1. Upper Bounded Wildcard (`? extends T`)

Ye wildcard restrict karta hai ki type ya toh **T** hoga ya phir **T ki koi child class**.

- **Matlab:** "Mujhe T ya usse chota (subclass) koi bhi chalega."
- **Purpose (Get):** Ye data **read** karne ke liye best hai. Aapko guarantee milti hai ki jo bhi bahar aayega wo `T` type ka hoga.
- **Restriction:** Aap isme naya data **add nahi kar sakte** (sirf `null` add kar sakte hain), kyunki compiler ko confirm nahi hota ki list `Integer` ki hai, `Double` ki hai ya kisi aur subclass ki.

2. Lower Bounded Wildcard (`? super T`)

Ye wildcard restrict karta hai ki type ya toh **T** hoga ya phir **T ki koi parent class**.

- **Matlab:** "Mujhe T ya usse bada (superclass) koi bhi chalega."
- **Purpose (Put):** Ye data **write/add** karne ke liye best hai. Aap isme `T` type ke objects safe tareeke se daal sakte hain.
- **Restriction:** Isse data **read** karna mushkil hota hai kyunki return type hamesha `Object` milega (sabse bada parent).

Asal mein, `T` ek **Variable (Naam)** hai aur `?` ek **Unknown (Anjaan)** hai.

---

1. Ek Story se samjho (The "Marriage" Example)

Case A: `<T>` (Type Parameter)

Maano ek shaadi ho rahi hai. Rule ye hai: **"Jo (T) dulhan ke ghar wale khayenge, wahi (T) dulhe ke ghar wale khayenge."**

- Agar dulhan walo ne *Paneer* khaya (T = Paneer), toh dulhe walo ko bhi *Paneer* hi milega.
- Aapne ek **Link** bana diya. `T` ne dono side ko baandh diya.

Case B: `<?>` (Wildcard)

Ab doosra rule dekho: **"Is counter par koi bhi (?) aa kar kha sakta hai."**

- Yahan koi link nahi hai. Pehla banda *Ice-cream* kha sakta hai, doosra *Samosa*.
- Humein matlab nahi hai ki kaun aa raha hai, bas "khana" hona chahiye.

Wildcard ke 3 types (Asaan Examples ke saath)

1. Unbounded Wildcard (`<?>`)

Iska matlab hai: **"Kuch bhi chalega!"** Aapko bas data se matlab hai, uske type se nahi.

- **Example:** Ek method jo kisi bhi list ka size bataye ya use print kare.
- **Code:** `void printList(List<?> list) { ... }`
- **Fayda:** Aap isme `List<String>`, `List<Integer>`, ya `List<Dog>` kuch bhi bhej sakte hain.

2. Upper Bounded Wildcard (`<? extends T>`)

Iska matlab hai: **"T ya uske niche waale (Subclasses) hi chalenge."** Ye hum tab use karte hain jab humein data **Read** karna ho.

- **Example:** Ek method jo sirf Numbers ko add kare.
- **Code:** `void sum(List<? extends Number> list) { ... }`
- **Kaise kaam karega:** Aap `List<Integer>` ya `List<Double>` bhej sakte hain kyunki dono `Number` ke andar aate hain. Lekin `List<String>` nahi bhej sakte.

3. Lower Bounded Wildcard (`<? super T>`)

Iska matlab hai: **"T ya uske upar waale (Parent classes) hi chalenge."** Ye hum tab use karte hain jab humein list mein data **Write (Add)** karna ho.

- **Example:** Agar aapko ek list mein `Integer` daalna hai, toh wo list ya toh `Integer` ki honi chahiye, ya `Number` ki, ya phir `Object` ki.
- **Code:** `void addNumber(List<? super Integer> list) { ... }`

Example:

1. Type Parameter (`<T>`) ka Example

Jab humein **ek jaisa type** fix karna ho.

Java

```
// Yahan 'T' ek naam hai. Jo type aayega, wahi return hoga.
public <T> T getFirstElement(List<T> list) {
    return list.get(0);
}

// Use:
List<String> names = List.of("Ram", "Shyam");
String name = getFirstElement(names); // T yahan 'String' ban gaya
```

2. Wildcard (`?`) ka Example

Jab humein type se koi matlab nahi, bas flexibility chahiye.

Upper Bound (`? extends Fruit`) - Sirf Read ke liye

"Mujhe aisi list do jisme ya toh `Fruit` ho ya `Apple` . Main use kha (read) sakta hoon."

Java

```
public void eatFruits(List<? extends Fruit> list) {
    for (Fruit f : list) {
        System.out.println("Eating..."); // Hum read kar pa rahe hain
    }
    // list.add(new Apple()); // ERROR! Compiler ko nahi pata list Apple ki hai y
}
```

Lower Bound (`? super Apple`) - Sirf Write ke liye

"Mujhe aisi list do jisme main `Apple` **daal** sakun. Wo list `Apple` ki bhi ho sakti hai aur `Fruit` ki bhi."

Java

```
public void addApple(List<? super Apple> list) {
    list.add(new Apple()); // SUCCESS! Hum data daal pa rahe hain.
}
```

Inner class and nested Static Class in Java:

Where should you use nested class in Java? This is one of the tricky Java question, If a class can only be useful to one particular class, it make sense to keep that inside the class itself otherwise declare it as top level class. Nested class improves Encapsulation and help in maintenance. I actually look JDK itself for examples and if you look HashMap class, you will find that Map.Entry is a good example of nested class in Java.

Inner Class:

Any class which is not a top level or declared inside another class is known as nested class and out of those nested classes, class which are declared non static are known as Inner class in Java. there are three kinds of Inner class in Java:

1. Local inner class
2. Anonymous inner class
3. Member inner class

Local inner class is declared inside a code block or method. Anonymous inner class is a class which doesn't have name to reference and initialized at same place where it gets created. Member inner class is declared as non static member of outer class.

Here are some important properties of Inner classes in Java:

1) In order to create instance of Inner class, an instance of outer class is required. Every instance of inner class is bounded to one instance of Outer class.

2) Member inner class can be make private, protected or public. its just like any other member of class.

3) Local inner class can not be private, protected or public because they exist only inside of local block or method. You can only use final modifier with local inner class.

4) Anonymous Inner class are common to
implement Runnable or CommandListener where you just need to implement one method. They are created and initialized at same line.

5) You can access current instance of Outer class, inside inner class
as Outer.this variable.

Serializable:

If you want a class object to be serializable, all you need to do it implement the java.io.Serializable interface. Serializable in java is a marker interface and has no fields or methods to implement. It's like an Opt-In process through which we make our classes serializable.

Serialization in java is implemented by ObjectInputStream and ObjectOutputStream, so all we need is a wrapper over them to either save it to file or send it over the network.

Object serialization is the mechanism of converting object into **byte stream.** Where as the reverse process of recreating the object from those serialized bytes is known as **deserialization.**

Example:

Address.java:

```
public class Address implements Serializable{

	private static final long serialVersionUID = -7800233291437691948L;
	private int id;
	private String addressLine1;
	private String addressLine2;
	private String city;
	private long zipCode;

	public Address(int id, String addressLine1, String addressLine2, String city, long zipCode) {
		this.id = id;
		this.addressLine1 = addressLine1;
		this.addressLine2 = addressLine2;
		this.city = city;
		this.zipCode = zipCode;
	}
```

//Setter and getters

Employee Class:

```java
public class Employee implements Serializable {

	private static final long serialVersionUID = -4724957588046435474L;
	private int id;
	private String name;
	private int age;
	private String email;
	private String password;

	private Address address;

	public Employee(int id, String name, int age, String email, String password, Address address) {
		System.out.println( "Employee args constructor is called..");
		this.id = id;
		this.name = name;
		this.age = age;
		this.email = email;
		this.password = password;
		this.address = address;
	}
```

Mail Class:

```
public class ClientTest {

	public static void main(String[] args) {

		String fileName = "employee.ser";
		//serializeObject(fileName);
		deserializeObject(fileName);

	}

	private static void deserializeObject(String fileName) {
		try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File(fileName)))) {
			Object object = ois.readObject();
			Employee employee=(Employee)object;
			System.out.println(employee);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	private static void serializeObject(String fileName) {
		Address address = new Address(1111, "address Line1", "address Line2", "Mumbai", 590999);
		Employee employee = new Employee(1001, "KK", 30, "kishan.javatrainer@gmail.com", "pass@123",address);

		try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File(fileName)))) {
			oos.writeObject(employee);
			System.out.println("Object is serialized..");
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
```

Example #2:

Write method -> writeObject() and readObject()

Sometimes we need to extend a class that doesn't implement Serializable interface. If we rely on the automatic serialization behavior and the superclass has some state, then they will not be converted to stream and hence not retrieved later on.

This is one place, where readObject() and writeObject() methods really help. By providing their implementation, we can save the super class state to the stream and then retrieve it later on. Let's see this in action.

Model class Person with setter getter and Main class like below:

```
public class Employee extends Person implements Serializable {

	private static final long serialVersionUID = -4724957588046435474L;

	private String email;
	private String password;

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	private void writeObject(ObjectOutputStream oos) throws IOException {
		// calling default functionality for Employee fields
		oos.defaultWriteObject();

		// Explicitly writing Person fields
		oos.writeInt(getId());
		oos.writeObject(getName());
		oos.writeInt(getAge());
	}
private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException {
		 // calling default functionality for Employee fields
		ois.defaultReadObject();

		// Explicitly reading Person fields and setting them
		setId(ois.readInt());
		setName((String)ois.readObject());
		setAge(ois.readInt());
}
@Override
public String toString() {
		return "Employee [email=" + email + ", password=" + password + "]";
}
```

Structure should be same

```java
public class ClientTest {

	public static void main(String[] args) {

		String fileName = "employee.ser";
		serializeObject(fileName);
		deserializeObject(fileName);

	}

	private static void deserializeObject(String fileName) {
		try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File(fileName)))) {
			System.out.println("Object is deserialized..");
			Object object = ois.readObject();
			Employee employee=(Employee)object;
			System.out.println("ID:"+employee.getId());
			System.out.println("Name:"+employee.getName());
			System.out.println("Age:"+employee.getAge());
			System.out.println("Email:"+employee.getEmail());
			System.out.println("Password:"+employee.getPassword());
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
	private static void serializeObject(String fileName) {
		Employee employee = new Employee();
		employee.setId(1001);
		employee.setName("KK");
		employee.setAge(30);

		employee.setEmail("kishan.javatrainer@gmail.com");
		employee.setPassword("pass@123");

		try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File(fileName)))) {
			oos.writeObject(employee);
			System.out.println("Object is serialized..");
			System.out.println("-------------------------------------");
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

}
```

What would happen if the SerialVersionUID of an object is not defined?

If you don't define serialVersionUID in your serilizable class, Java compiler will make one by creating a hash code using most of your class attributes and features. When an object gets serialized, this hash code is stamped on the object which is known as the SerialVersionUID of that object. This ID is required for the version control of an object. SerialVersionUID can be specified in the class file also. In case, this ID is not specified by you, then Java compiler will regenerate a SerialVersionUID based on updated class and it will not be possible for the already serialized class to recover when a class field is added or modified. Its recommended that you always declare a serialVersionUID in your Serializable classes.

Externalizable:

Externalizable interface, Serializable ka hi ek advanced version hai. Ye aapko serialization process par **poora control** deta hai.

Jab aap Serializable use karte hain, toh Java khud hi objects ko save aur restore karta hai (automatic). Lekin Externalizable mein **aapko** batana padta hai ki kaun sa data save karna hai aur kaun sa nahi.

Key Features

- **Package:** `java.io.Externalizable`
- **Child Interface:** Ye `Serializable` interface ko hi extend karta hai.
- **Methods:** Isme do compulsory methods hote hain:
  1. `writeExternal(ObjectOutput out)` : Object ko save karne ke liye.
  2. `readExternal(ObjectInput in)` : Object ko restore karne ke liye.

Externalizable vs Serializable

Feature	Serializable	Externalizable
Control	Java control karta hai (Automatic).	Developer control karta hai (Manual).
Performance	Slow (Reflection use hota hai).	Fast (Custom logic hota hai).
Methods	Koi method nahi (Marker Interface).	Do methods: `writeExternal` aur `readExternal`.
Constructor	Constructor ki zaroorat nahi hoti.	**Public No-arg Constructor** compulsory hai.
Transient Keyword	`transient` se data hide hota hai.	`transient` ka koi asar nahi hota (aap khud decide karte hain kya likhna hai).

If you notice the java serialization process, it's done automatically. Sometimes we want to obscure the object data to maintain it's integrity. We can do this by implementing `java.io.Externalizable` interface and provide implementation of *writeExternal()* and *readExternal()* methods to be used in serialization process.

Example: **Person class not implementing externalization.**

```
public class Person {

	private int id;
	private String name;
	private int age;
```

Setter and Getter

Employee Class:

```java
public class Employee extends Person implements Externalizable {

	public Employee() {
		System.out.println("Employee default constructor is called..");
	}

	private String email;
	private String password;

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	@Override
	public void writeExternal(ObjectOutput out) throws IOException {
		//Writing parent class ClassA fields
		out.writeInt(getId());
		out.writeObject(getName());
		out.writeInt(getAge());

		// Writing child class fields
		out.writeObject(getEmail());
		out.writeObject(getPassword());
	}

	@Override
	public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
		// Setting parent class fields
			setId(in.readInt());
			setName((String)in.readObject());
			setAge(in.readInt());

			// Setting child class fields
			setEmail((String)in.readObject());
			setPassword((String)in.readObject());
	}
}
```

SerialVersionUID:

SerialVersionUID Java Serialization ka ek bohot hi mahatvapurn concept hai. Ye ek **unique identifier** (ek `long` number) hota hai jo har `Serializable` class ke saath juda hota hai.

Iska main kaam ye check karna hai ki **Sender** (jo object save kar raha hai) aur **Receiver** (jo object load kar raha hai) dono ek hi version ki class use kar rahe hain ya nahi.

Ye kaam kaise karta hai?

Jab aap kisi object ko serialize karte hain, toh Java runtime us class ka ek version number (SerialVersionUID) us serialized file ke saath attach kar deta hai.

1. **Deserialization ke waqt:** Java check karta hai ki file mein jo ID hai aur aapki current class ki jo ID hai, kya wo **match** kar rahi hain?
2. **Agar match hoti hai:** Toh object successfully restore ho jata hai.
3. **Agar match nahi hoti:** Toh Java `InvalidClassException` throw kar deta hai.

Isse declare karne ka tarika

Aap ise apni class mein is tarah define karte hain:

Java

```java
public class MyClass implements Serializable {
    // Unique ID declare karna
    private static final long serialVersionUID = 1L;

    private String name;
    private int id;
}
```

Kya hota hai agar aap ise define NA karein?

Agar aap manually `serialVersionUID` nahi likhte, toh Java compiler **automatic** ek ID generate karta hai jo class ke variables, methods, aur modifiers par depend karti hai.

Iska nuksan (The Problem): Maan lijiye aapne object save kiya, aur phir class mein ek chota sa change kar diya (jaise ek naya variable add karna). Ab agar aap purane save kiye huye object ko load karenge, toh Java ki automatic generate ki hui ID badal jayegi, aur aapka purana data load nahi ho payega—bhale hi wo change minor ho.

Important Point

Agar aapne apni class mein koi **breaking change** kiya hai (jaise kisi variable ka data type `int` se `String` kar diya), toh aapko **SerialVersionUID ko badal dena chahiye** (`1L` se `2L`). Isse purana incompatible data load nahi hoga aur system safe rahega.

Kya aap janna chahte hain ki IDEs (jaise IntelliJ ya Eclipse) is ID ko automatically kaise generate karte hain?

Enum:

1. Enum (Keyword/Data Type)

`enum` (Enumeration ka short form) Java 5 mein introduce kiya gaya tha. Ye ek **special data type** hai jiska use **fixed constants** (niyat maan) ko define karne ke liye hota hai.

- **Kab use karein:** Jab aapko pata ho ki values fixed hain (e.g., Days of week, Directions, Order Status).
- **Khaas Baat:** Ye internally ek `class` hoti hai, isliye ismein methods aur constructors bhi ho sakte hain.

```java
package springframework.guru.enumexample;

import java.util.ArrayList;

public class EnumIteration {
    enum Days{
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
    }

    public ArrayList<String> enumIteration() {
        Days[] days = Days.values();
        ArrayList<String> stringDay = new ArrayList<String>();
        for (Days day : days) {
            stringDay.add(day.toString());
        }
        return stringDay;
    }
}
```

Key Points

Some key points on the enum type are:

- Java enums extend the java.lang.Enum class implicitly. Therefore, you cannot extend any other class in enum.
- Enums can implement interfaces. They implicitly implement the Serializable and Comparable interfaces. This means if you need some common functionality along diverse enum types, you can define it with an interface to have the enums provide the method implementations.
- In Java, enum is a keyword. Enum constructors are always private or default. Therefore, you cannot have public or protected constructors in an enum type.
- In an enum definition, comma separated constants must be terminated with a semicolon.
- You cannot create an instance of enum using the new operator.
- You can declare abstract methods within an enum. If you do so, all the enum fields must implement the abstract methods.
- You can use the "==" operator to compare enum constants effectively, since enum constants are final.

Enumeration (Interface):

`Enumeration<E>` ek **legacy interface** hai jo `java.util` package mein hota hai. Iska use collection (jaise Vector ya HashTable) ke elements ko **iterate** (ek-ek karke nikalne) ke liye kiya jata hai.

- **Kab use karein:** Ye purane Java versions mein use hota tha. Aaj kal iski jagah `Iterator` ne le li hai.
- **Methods:** Ismein do main methods hote hain: `hasMoreElements()` aur `nextElement()`.

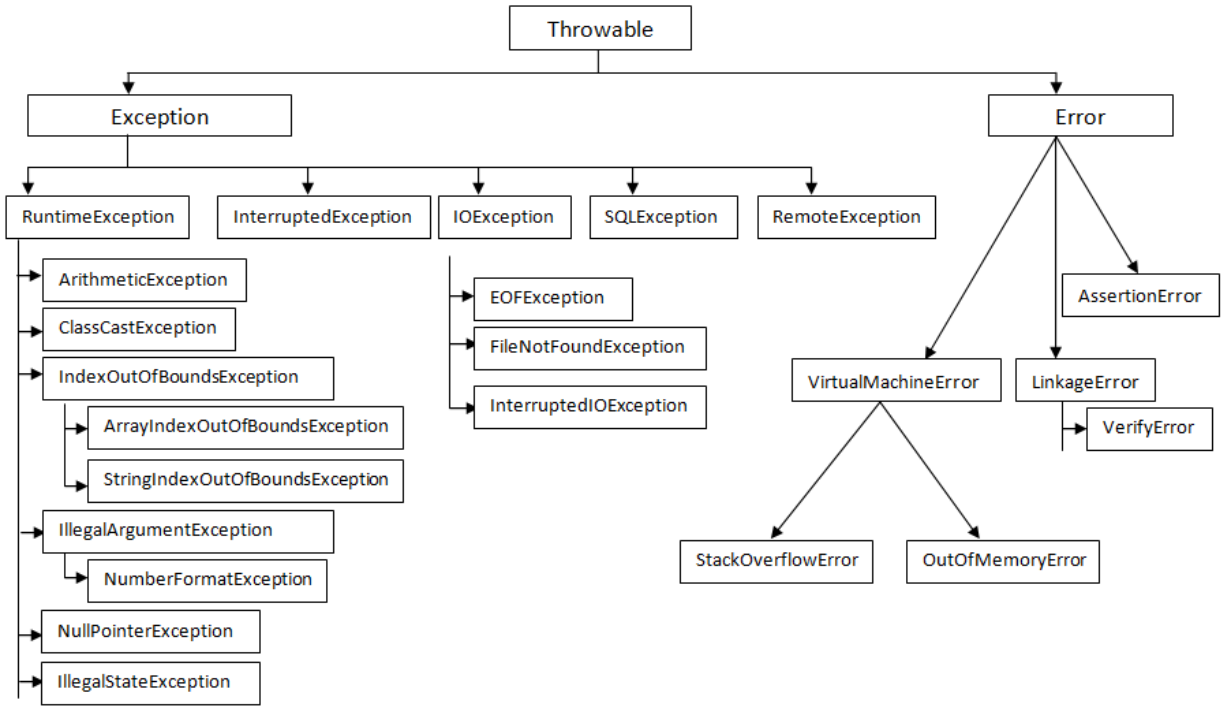
Java

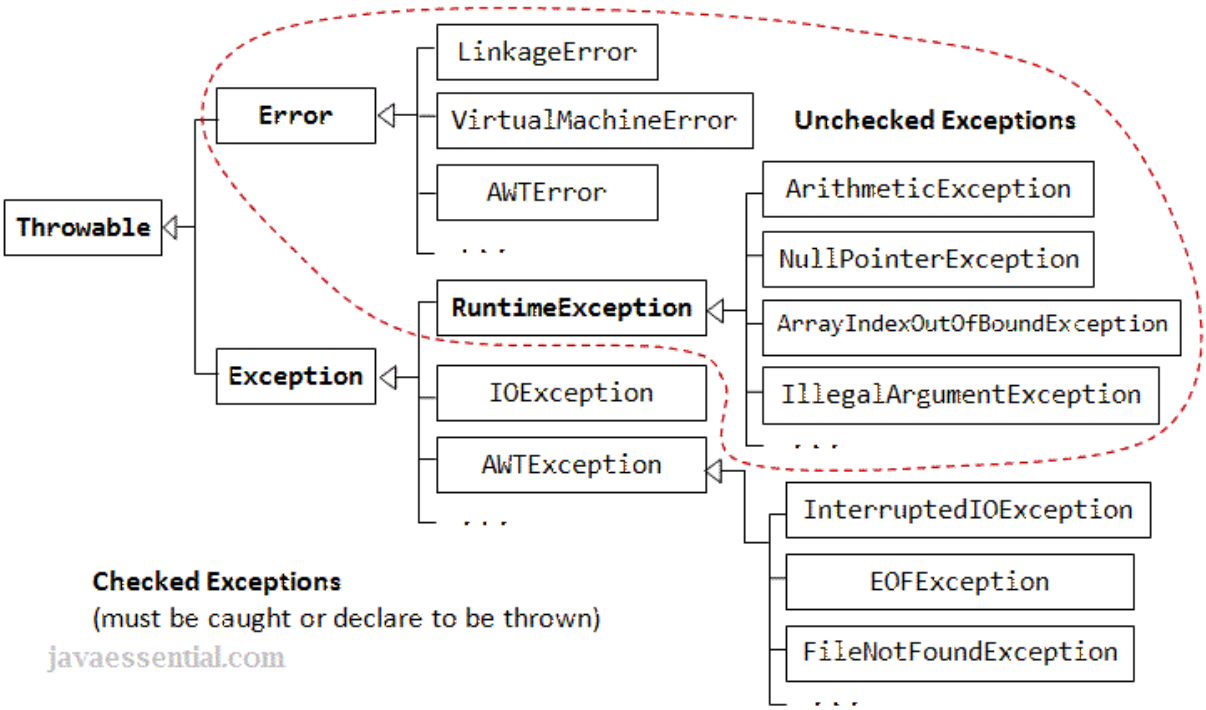
```
Vector<String> v = new Vector<>();
v.add("Java");
Enumeration<String> e = v.elements();
while(e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

Enumeration Vs Iterator In Java :

Enumeration	**Iterator**
Using *Enumeration*, you can only traverse the collection. You can't do any modifications to collection while traversing it.	Using *Iterator*, you can remove an element of the collection while traversing it.
Enumeration is introduced in JDK 1.0	*Iterator* is introduced from JDK 1.2
Enumeration is used to traverse the legacy classes like *Vector*, *Stack* and *HashTable*.	*Iterator* is used to iterate most of the classes in the collection framework like *ArrayList*, *HashSet*, *HashMap*, *LinkedList* etc.
Methods : *hasMoreElements()* and *nextElement()*	Methods : *hasNext()*, *next()* and *remove()*
Enumeration is fail-safe in nature.	*Iterator* is fail-fast in nature.
Enumeration is not safe and secured due to it's fail-safe nature.	*Iterator* is safer and secured than *Enumeration*.

Exception Handling:





Exception & Error:

Exception	Error
1) Exceptions can be recovered	1) Errors cannot be recovered.
2) Exceptions can be classified in to two types: a) Checked Exception b) Unchecked Exception	2) There is no such classification for errors. Errors are always unchecked.
3) In case of checked Exceptions compiler will have knowledge of checked exceptions and force to keep try catch blocks.	3) In case of Errors compiler won't have knowledge of errors.

fully checked exception & Partially checked exception:

* **A Checked Exception is said to be fully checked exception if and only if all its child classes also checked.**

Example:

IOException

InteruptedException

* **A Checked Exception is said to be partially Checked exception if and only if some of its child classes are Unchecked.**

Example:

Exception

Note:

The only possible partially checked exceptions in java are

1. Exception
2. Throwable

final, finally & Finalize() :

final-:

1. final is an modifier applicable for classes,methods and variables. If a class declare as final then we can not extends that class.i.e., we cannot create child class for that class.

2. If a method declare as a final then we cannot override that method in the child class.

3. If a variable declare as a final then it will become constant and can't perform re-assignment for that variable.

finally-:

finally is a block always associated with try catch to maintain "CleanUp" code.

Finalize()-:

Finalize() is a method which is always invoked by "Garbage Collector" just before destroying an object to perform cleanup activities.

Note:

- Finally meant for cleanup activities related to try block. Where as finalize() meant for cleanup activities related to object.

throw & throws:

throw	throws
Java **throw** keyword is used to explicitly throw an exception.	Java **throws** keyword is used to declare an exception.
Checked exception cannot be propagated using **throw**.	Checked exception can be propagated with **throws**.
If we see syntax wise, **throw** is followed by an instanceof Exception class Example : throw new NumberFormatException("The month entered, is invalid.");	If we see syntax wise, **throws** is followed by exception class names. Example : throws IOException,SQLException
The keyword **throw** is used inside method body.	**throws clause** is used in method declaration (signature).
By using **throw keyword** in java you cannot throw more than one exception. Example: throw new IOException("Connection failed!!")	By using **throws** you can declare multiple exceptions. Example: public void method() throws IOException,SQLException.

ClassNotFoundException & NoClassDefFoundError

Difference	ClassNotFoundException	NoClassDefFoundError
Definition	At the runtime if the corresponding .class file not there then we will get "**ClassNotFoundException**. If we not set hardcoded class name.	At the runtime if **.class** file is not there to the corresponding .java file then we will get "**NoClassDefFoundError**" If we hardcoded uses the class then only we well get this error.
Example	Object obj=Class.forName("Test").newInstance(); If at the runtime Test.class not then we will get this exception. Note : For Dynamically provided hardcoded class name, at runtime .class file not there then we will get this exception.	Test t=new Test(); If at the runtime Test.class not then we will get this exception. Note: For hardcoded class name at the runtime .class file not there then only we wil get this error.
Which exception	Checked Exceptions	Unchecked Exception

CheckedException & UncheckedException:

Difference	CheckedException	UncheckedException

Definition	The exception which are checked by compiler for smooth execution of the program at runtime are called checkedexception.
	Example :
	HallticketMissingException
	PenNotWorkingException
	FileNotFoundException
	
	In the case of checked exceptions compiler will check whether we are handling exception if the programmer not handling then we will get compile time error.
	
	Public static void main(String[] arge){
	PrintWriter pw=new
	PrintWriter("abc.txt);
	pw.write("hello");
	
	}
	Compiler will tell you :
	Exception: Unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown.
		RuntimeException and it's child classes, Error and it's child classes are unchecked. Except this remaining are checked exceptions.
		Example:
		Arithmetic Exception
		System.out.println(10/0);

	Checked by compiler time	Checked by runtime
Extends	Exception	RuntimeException
	Show recoverable condition	Show programming error

Collections Framework:

Java **Collection Framework** ek single architecture hai jo objects ke group ko store aur manipulate karne ke liye use hota hai.

Pehle (JDK 1.2 se pehle), Java mein data store karne ke liye alag-alag tarike the jaise Vector, Hashtable, aur Array. In sabka apna alag style tha. Collection Framework ne in sabko ek standard interface ke niche la kar kaam aasan kar diya.

Collection Hierarchy (Strucutre)

Collection Framework mainly do parts mein divided hai:

1. **java.util.Collection:** Isme `List`, `Set`, aur `Queue` aate hain.
2. **java.util.Map:** Ye key-value pairs ke liye hota hai (ye technically `Collection` interface ko extend nahi karta, par framework ka hissa hai).

1. List (Ordered Collection)

Ismein elements ka sequence maintain rehta hai aur **duplicates** allow hote hain.

- **ArrayList:** Fast retrieval (index based), slow manipulation (delete/insert).
- **LinkedList:** Fast manipulation, slow retrieval.
- **Vector:** Thread-safe version (legacy class).

2. Set (Unique Elements)

Ismein **duplicates** allowed nahi hote.

- **HashSet:** No order maintained, fast performance.
- **LinkedHashSet:** Insertion order maintain karta hai.
- **TreeSet:** Elements ko sorted order (A-Z, 1-10) mein rakhta hai.

3. Queue (First In First Out)

Elements ko processing ke liye line mein lagane ke liye.

- **PriorityQueue:** Priority ke hisaab se elements process karta hai.
- **Deque (ArrayDeque):** Dono ends se insert/delete kar sakte hain.

4. Map (Key-Value Pair)

Har element ki ek unique key hoti hai.

- **HashMap:** No order, fast operations.
- **LinkedHashMap:** Insertion order maintain karta hai.
- **TreeMap:** Keys ko sorted order mein rakhta hai.

Comparable & Comparator Interfaces:

```
package java.lang;

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

```
package java.util;

public abstract interface Comparator<T>
{
  public abstract int compare(T paramT1, T paramT2);

  public abstract boolean equals(Object paramObject);
}
```

Comparable Interface Example:

```java
public class Laptop implements Comparable<Laptop>{
private String company;
private int price;

public Laptop(String company, int price) {
    super();
    this.company = company;
    this.price = price;
}
public String getCompany() {
    return company;
}
public void setCompany(String company) {
    this.company = company;
}
public int getPrice() {
    return price;
}
public void setPrice(int price) {
    this.price = price;
}

@Override
public int compareTo(Laptop lp) {
    // TODO Auto-generated method stub
    if(this.getPrice()>lp.getPrice()){
        return 1;
    }else if(this.getPrice()<lp.getPrice()){
        return -1;
    }
    return 0;

}

}
```

```java
package com.example;

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

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
     List<Laptop> list=new ArrayList<Laptop>();
     Laptop laptop1=new Laptop("Apple", 200000);
     Laptop laptop2=new Laptop("Chrome", 100000);
     Laptop laptop3=new Laptop("Lenovo", 26000);

     list.add(laptop1);
     list.add(laptop2);
     list.add(laptop3);

     Collections.sort(list);

     for(Laptop laptop:list)
     System.out.println(laptop.getPrice());

    }
}

```

```
<terminated> Main (15) [Java Applica
26000
100000
200000
```

Comparator Interface Example:

```java
package com.example;

public class Laptop{
private String company;
private int price;

public Laptop(String company, int price) {
    super();
    this.company = company;
    this.price = price;
}
public String getCompany() {
    return company;
}
public void setCompany(String company) {
    this.company = company;
}
public int getPrice() {
    return price;
}
public void setPrice(int price) {
    this.price = price;
}
}

```

Main.java | Laptop.java

```
import java.util.List;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
     List<Laptop> list=new ArrayList<Laptop>();
     Laptop laptop1=new Laptop("Apple", 200000);
     Laptop laptop2=new Laptop("Chrome", 100000);
     Laptop laptop3=new Laptop("Lenovo", 26000);

     list.add(laptop1);
     list.add(laptop2);
     list.add(laptop3);

     Collections.sort(list,new Comparator<Laptop>() {

        @Override
        public int compare(Laptop lp1, Laptop lp2) {
            // TODO Auto-generated method stub
            if(lp1.getPrice()>lp2.getPrice()){
                return 1;
            }else if(lp1.getPrice()<lp2.getPrice()){
                return -1;
            }
            return 0;
        }
    });
```

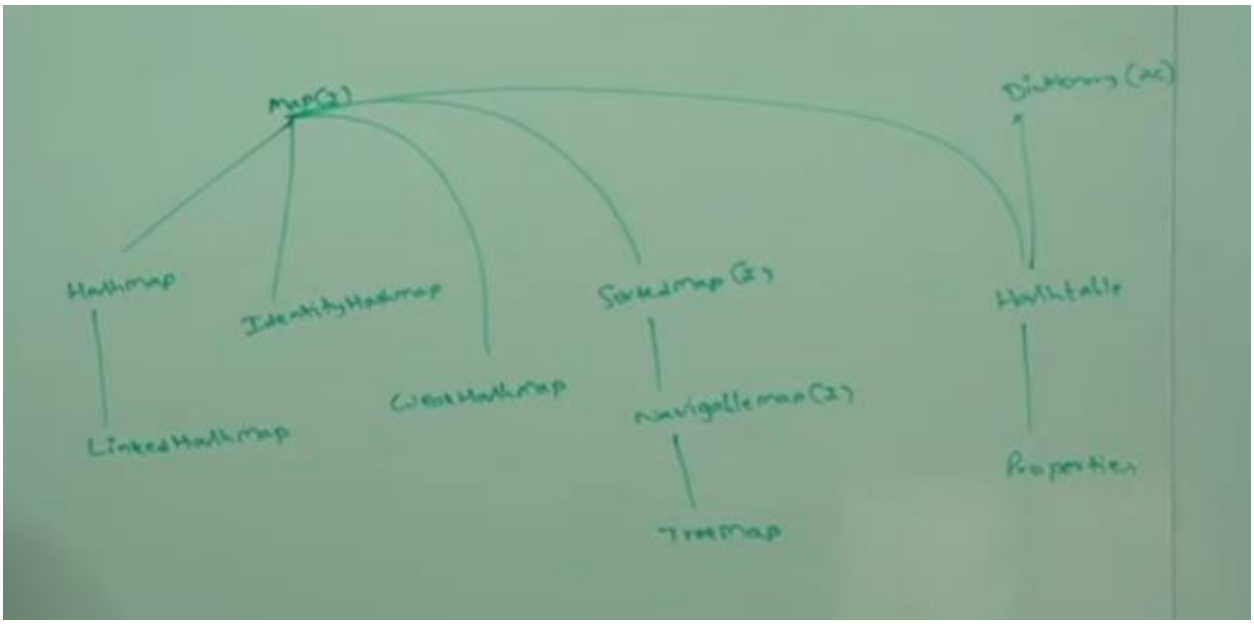
Console

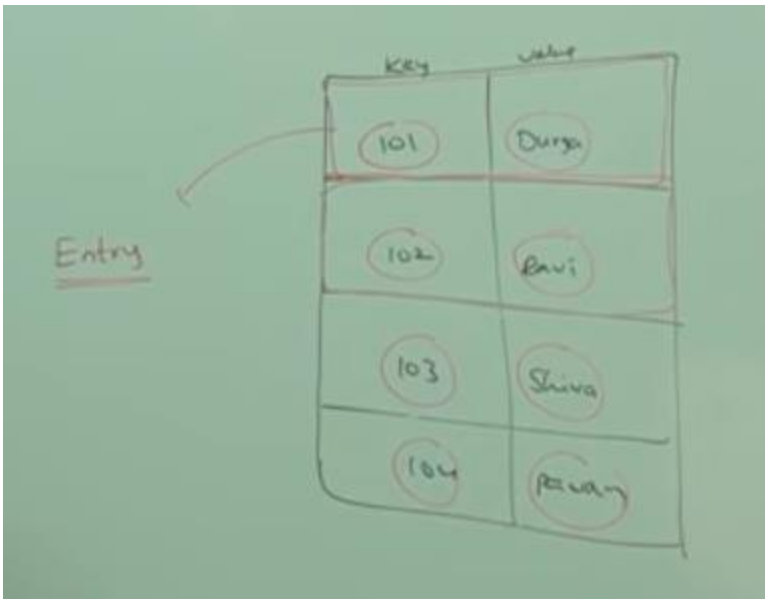
```
<terminated> Main (15) [
26000
100000
200000
```

Comparable	Comparator
1. It is meant for default natural sorting order.	1. It is meant for customized sorting order.
2. Present in java.lang package .	2. Present in java.util package
3. This interface defines only one method compareTo ().	3. This interface defines two methods compare() and equals().
4. All wrapper classes and String class implement comparable interface.	4. The only implemented classes of Comparator are Collator and RuleBasedCollator.

Comparable	Comparator
1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
2) Comparable **affects the original class** i.e. actual class is modified.	Comparator **doesn't affect the original class** i.e. actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is found in **java.lang** package.	Comparator is found in **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method.

Map Interface:



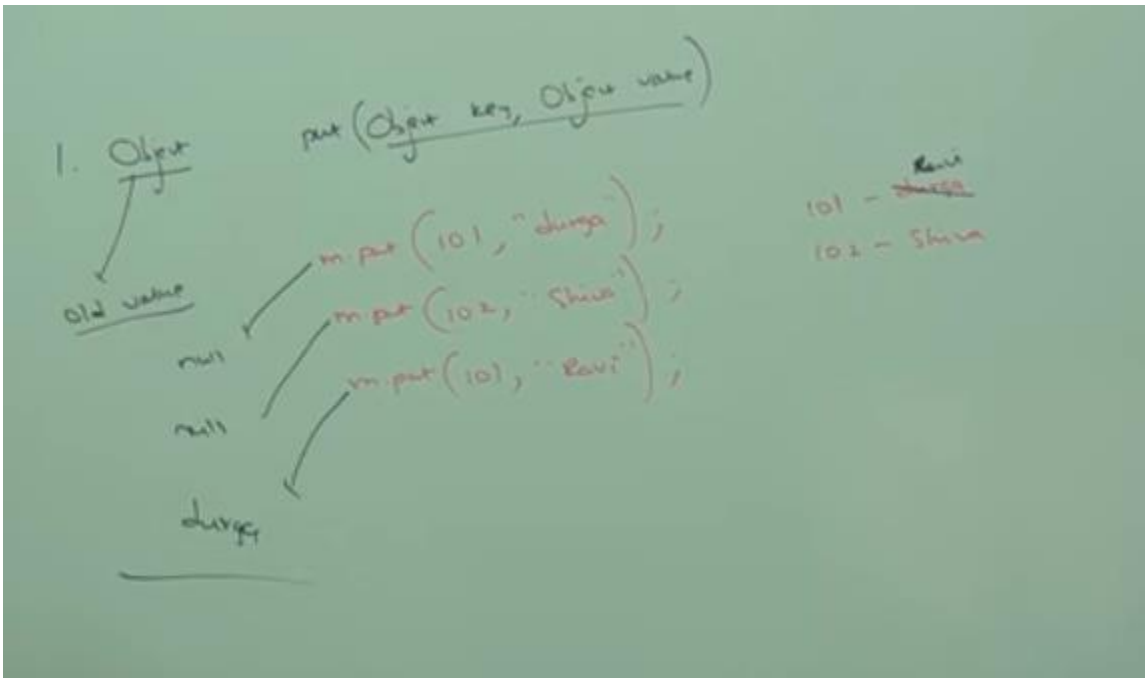


Map.Put() Method and return type:

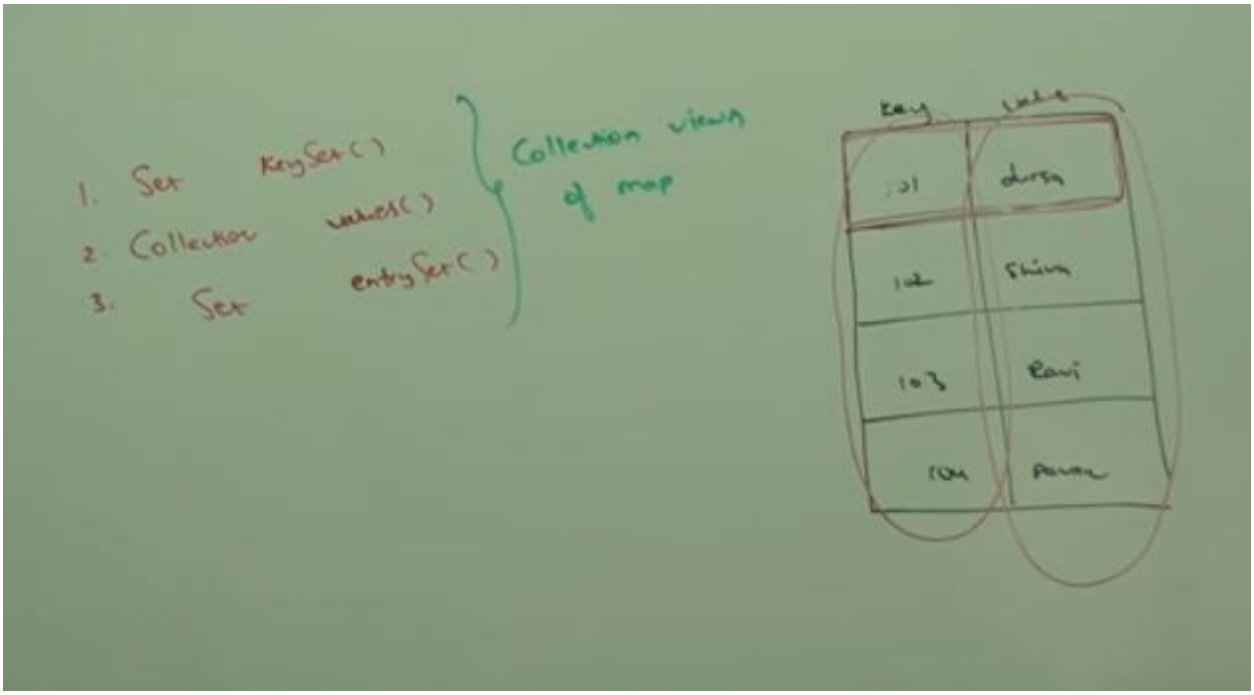
The return type for the put() method is object type.

If we add (map.put(101,'rahul')) then it will return previous value corresponding to the key(101).

Ex. If we are adding first time then it will return object with null value, but if second time we are adding (map.put(101,'BHATI')) then it will return object with "rahul" value.



Note:Collections Views Methods are below :



How hash map internally work?

In Hashmap collection class we are having put(key,value) and get(key) methods

By calling put(key,value) method, hasmap implementions calls hascode() method on key to get hashcode value ,which will get passed as an input to hashing function to get the bucket location to store the k-v pair in Map.Entry<K,V> in bucket.

Then at the time calling get(key) method ,even if 2 diffrents objects having same hashcode will get

fetched correctly by calling keys .equals() method to get the correct K,V pair to avoid collision.

Internal Working of HashMap in Java

In this article, we will see how hashmap's get and put method works internally. What operations are performed. How the hashing is done. How the value is fetched by key. How the key-value pair is stored.

As in previous article, HashMap contains an array of Node and Node can represent a class having following objects :

1. int hash
2. K key
3. V value
4. Node next

Now we will see how this works. First we will see the hashing process.

Hashing

Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap. Here I am taking key of my own class so that I can override hashCode() method to show different scenarios. My Key class is

```
//custom Key class to override hashCode()
// and equals() method
class Key
{
  String key;
  Key(String key)
  {
    this.key = key;
  }

  @Override
  public int hashCode()
  {
     return (int)key.charAt(0);
  }

  @Override
  public boolean equals(Object obj)
  {
    return key.equals((String)obj);
  }
}
```

Here overrided hashCode() method returns the first character's ASCII value as hash code. So whenever the first character of key is same, the hash code will be same. You should not approach this criteria in your program. It is just for demo purpose. As HashMap also allows null key, so hash code of null will always be 0.

hashCode() method

hashCode() method is used to get the hash Code of an object. hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object. It is possible to provide your own implementation of hashCode().

In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index.

equals() method

equals method is used to check that 2 objects are equal or not. This method is provided by Object class. You can override this in your class to provide your own implementation.

HashMap uses equals() to compare the key whether the are equal or not. If equals() method return true, they are equal otherwise not equal.

Buckets

A bucket is one element of HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. Buckets are different in capacity. A relation between bucket and capacity is as follows:

```
capacity = number of buckets * load factor
```

A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized.

Index Calculation in Hashmap

Hash code of key may be large enough to create an array. hash code generated may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So we generate index to minimize the size of array. Basically following operation is performed to calculate index.

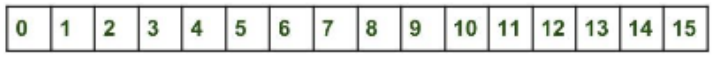
```
index = hashCode(key) & (n-1).
```

where n is number of buckets or the size of array. In our example, I will consider n as default size that is 16.

- **Initially Empty hashMap:** Here, the hashmap is size is taken as 16.

  ```
  HashMap map = new HashMap();
  ```

  HashMap :



- **Inserting Key-Value Pair:** Putting one key-value pair in above HashMap

```
map.put(new Key("vishal"), 20);
```

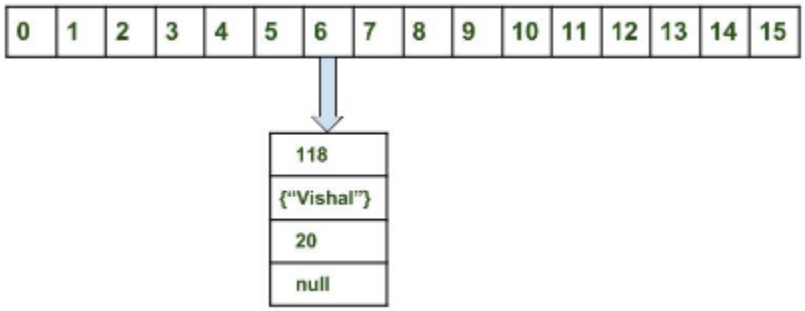
Steps:

1. Calculate hash code of Key {"vishal"}. It will be generated as 118.
2. Calculate index by using index method it will be 6.
3. Create a node object as :

```
{
  int hash = 118

  // {"vishal"} is not a string but
  // an object of class Key
  Key key = {"vishal"}

  Integer value = 20
  Node next = null
}
```

4. Place this object at index 6, if no other object is presented there.

Now HashMap becomes :



- **Inserting another Key-Value Pair:** Now, putting other pair that is,

```
map.put(new Key("sachin"),30);
```

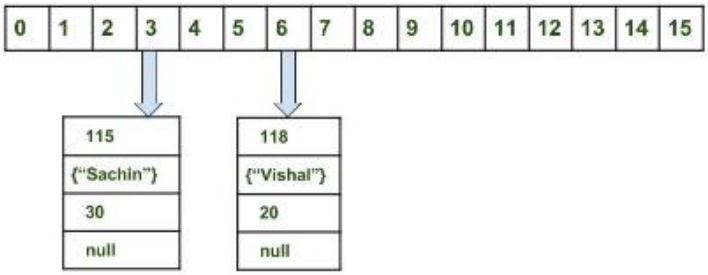
Steps:

1. Calculate hashCode of Key {"sachin"}. It will be generated as 115.
2. Calculate index by using index method it will be 3.
3. Create a node object as :

```
{
  int hash = 115
  Key key = {"sachin"}
  Integer value = 30
  Node next = null
}
```

Place this object at index 3 if no other object is presented there.

Now HashMap becomes :



- **In Case of collision:** Now, putting another pair that is,

```
map.put(new Key("vaibhav"),40);
```

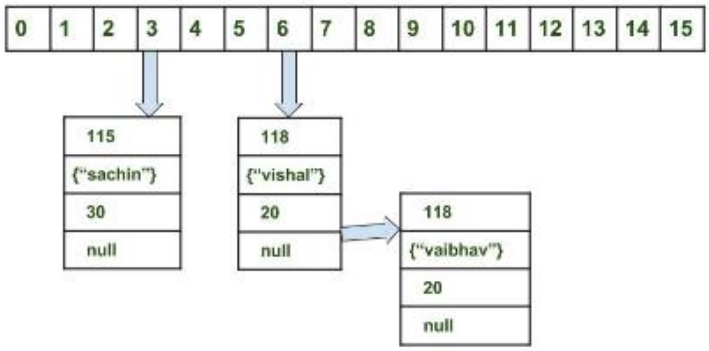
Steps:

1. Calculate hash code of Key {"vaibhav"}. It will be generated as 118.
2. Calculate index by using index method it will be 6.
3. Create a node object as :

```
{
 int hash = 118
 Key key = {"vaibhav"}
 Integer value = 40
 Node next = null
}
```

4. Place this object at index 6 if no other object is presented there.
5. In this case a node object is **found at the index 6** – this is a case of collision.
6. In that case, check via hashCode() and equals() method that if both the keys are same.
7. If keys are same, replace the value with current value.
8. Otherwise connect this node object to the previous node object via linked list and both are stored at index 6.

   Now HashMap becomes :



Using get method()

Now lets try some get method to get a value. get(K key) method is used to get a value by its key. If you don't know the key then it is not possible to fetch a value.

- **Fetch the data for key sachin:**

```
map.get(new Key("sachin"));
```

Steps:

1. Calculate hash code of Key {"sachin"}. It will be generated as 115.
2. Calculate index by using index method it will be 3.
3. Go to index 3 of array and compare first element's key with given key. If both are equals then return the value, otherwise check for next element if it exists.
4. In our case it is found as first element and returned value is 30.

- Fetch the data for key vaibahv:

```
map.get(new Key("vaibhav"));
```

Steps:

1. Calculate hash code of Key {"vaibhav"}. It will be generated as 118.
2. Calculate index by using index method it will be 6.
3. Go to index 6 of array and compare first element's key with given key. If both are equals then return the value, otherwise check for next element if it exists.
4. In our case it is not found as first element and next of node object is not null.
5. If next of node is null then return null.
6. If next of node is not null traverse to the second element and repeat the process 3 until key is not found or next is not null.

```
// Java program to illustrate
// internal working of HashMap
import java.util.HashMap;

class Key
{
    String key;
    Key(String key)
    {
        this.key = key;
    }

    @Override
    public int hashCode()
    {
        int hash = (int)key.charAt(0);
        System.out.println("hashCode for key: "
                           + key +" = "+ hash);
        return hash;
    }
```

```
    @Override
    public boolean equals(Object obj)
    {
        return key.equals(((Key)obj).key);
    }
}

// Driver class
public class GFG
{
    public static void main(String[] args)
    {
        HashMap map = new HashMap();
        map.put(new Key("vishal"), 20);
        map.put(new Key("sachin"), 30);
        map.put(new Key("vaibhav"), 40);

        System.out.println();
        System.out.println("Value for key sachin: " +
                           map.get(new Key("sachin")));
        System.out.println("Value for key vaibhav: " +
                           map.get(new Key("vaibhav")));
    }
}
```

Output:

```
hashCode for key: vishal = 118
hashCode for key: sachin = 115
hashCode for key: vaibhav = 118

hashCode for key: sachin = 115
Value for key sachin: 30
hashCode for key: vaibhav = 118
Value for key vaibhav: 40
```

Important Points

1. Time complexity is almost constant for put and get method until rehashing is not done.
2. In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
3. If key given already exist in HashMap, the value is replaced with new value.
4. hash code of null key is 0.
5. When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

Initial Capacity Of HashMap:

The capacity of an *HashMap* is the number of buckets in the hash table. The default initial capacity of the *HashMap* is 2^4 i.e 16. The capacity of the *HashMap* is doubled each time it reaches the threshold. i.e the capacity is increased to 2^5=32, 2^6=64, 2^7=128….. **when the threshold is reached.**

Load Factor Of HashMap :

Load factor is the measure which decides when to increase the capacity of the *HashMap*. The default load factor is 0.75f.

How The Threshold Is Calculated?

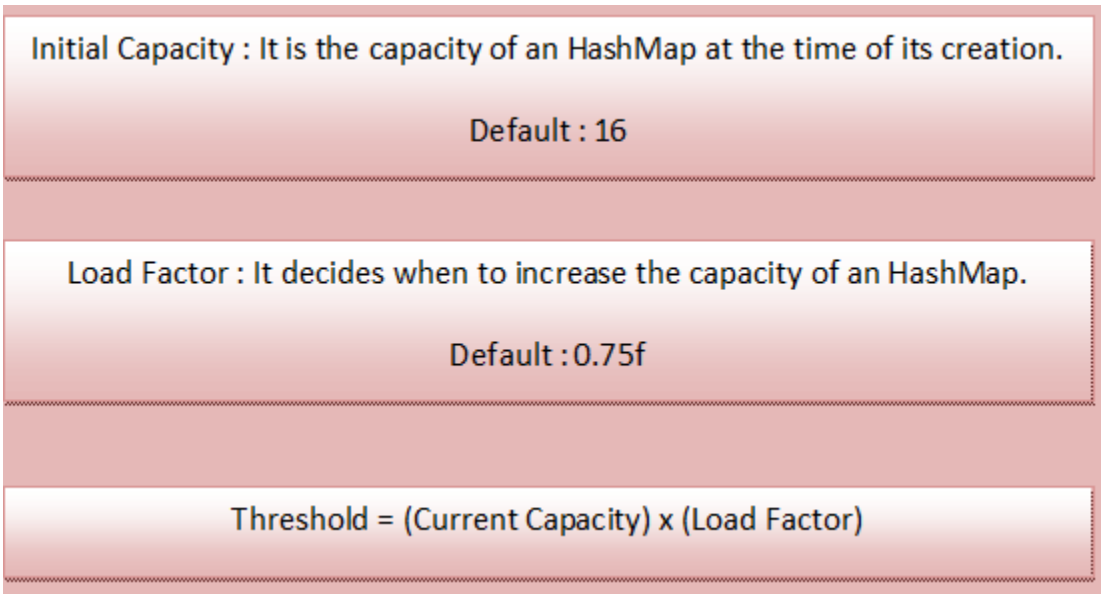
The threshold of an *HashMap* is the product of current capacity and load factor.

Threshold = (Current Capacity) * (Load Factor)

For example, if the *HashMap* is created with initial capacity of 16 and load factor of 0.75f, then threshold will be,

Threshold = 16 * 0.75 = 12

That means, the capacity of the *HashMap* is increased from 16 to 32 after the 12th element (key-value pair) is added into the *HashMap*.



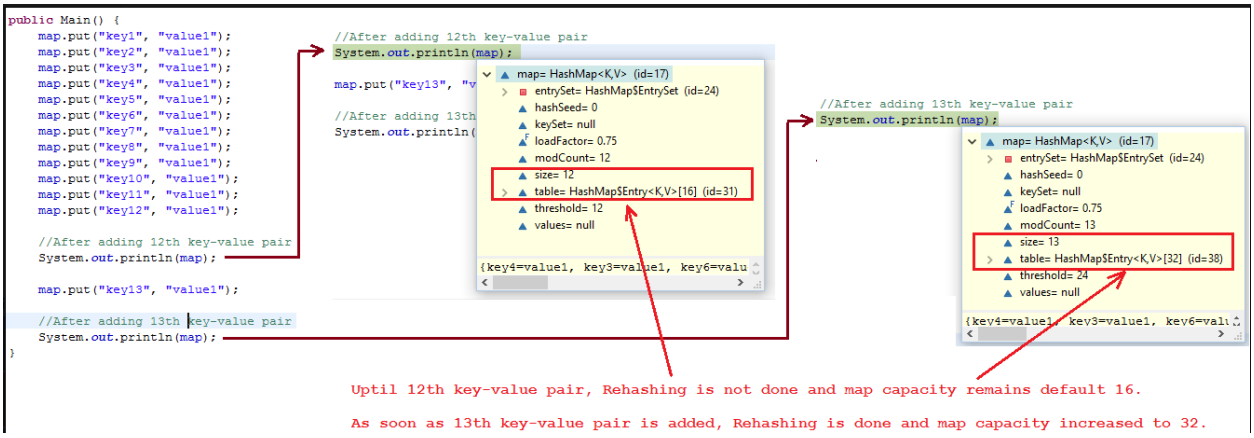
How Initial Capacity And Load Factor Affect Performance Of HashMap?

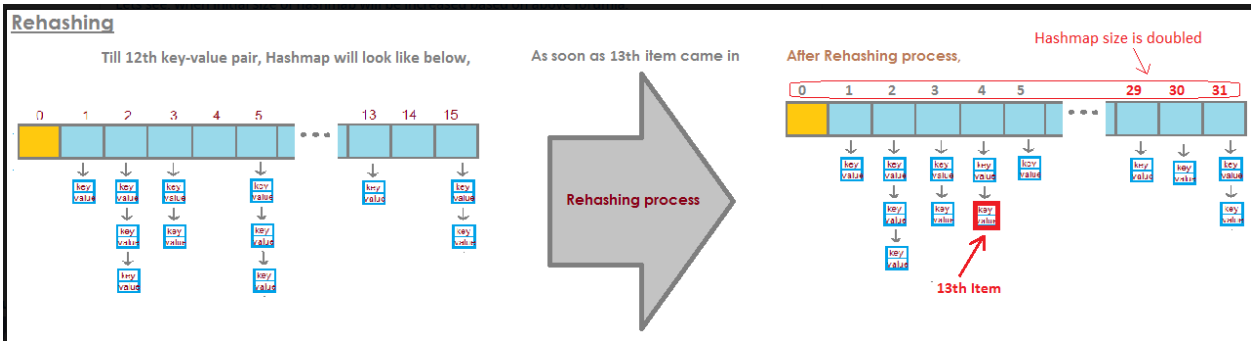
Whenever *HashMap* reaches its threshold, **rehashing** takes place. Rehashing is a process where new *HashMap* object with new capacity is created and all old elements (key-value pairs) are placed into new object after recalculating their hashcode. This process of rehashing is both space and time consuming. So, you must choose the initial capacity, by keeping the number of expected elements (key-value pairs) in mind, so that rehashing process doesn't occur too frequently.

You also have to be very careful while choosing the load factor. According to *HashMap* doc, the default load factor of 0.75f always gives best performance in terms of both space and time. For example,

If you choose load factor as 1.0f, then rehashing takes place after filling 100% of the current capacity. This may save the space but it will increase the retrieval time of existing elements. Suppose if you choose load factor as 0.5f, then rehashing takes place after filling 50% of the current capacity. This will increase the number of rehashing operations. This will further degrade the HashMap in terms of both space and time.

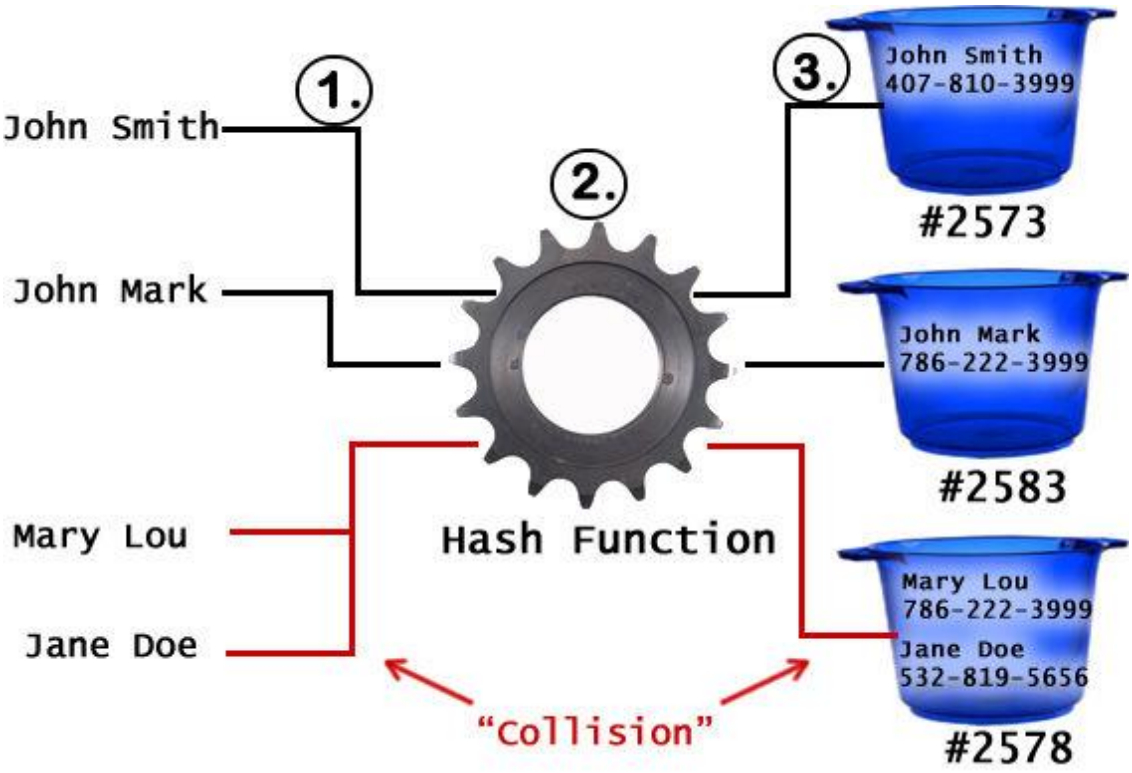
So, you have to be very careful while choosing the initial capacity and load factor of an *HashMap* object. Choose the initial capacity and load factor such that they minimize the number of rehashing operations.





Constructs an empty HashMap with the specified initial capacity and load factor.

```
Map<String, String> x = new HashMap<>(10, 0.85f);
```



HashMap work on Hashing Principle.

HashMap.Java

```
public V put(K paramK, V paramV)
{
  if (this.table == EMPTY_TABLE) {
    inflateTable(this.threshold);
  }
  if (paramK == null) {
    return (V)putForNullKey(paramV);
  }
  int i = hash(paramK);
  int j = indexFor(i, this.table.length);
  for (Entry localEntry = this.table[j]; localEntry != null; localEntry = localEntry.next)
  {
    Object localObject1;
    if ((localEntry.hash == i) && (((localObject1 = localEntry.key) == paramK) || (paramK.equals(localObject1))))
    {
      Object localObject2 = localEntry.value;
      localEntry.value = paramV;
      localEntry.recordAccess(this);
      return (V)localObject2;
    }
  }
  this.modCount += 1;
  addEntry(i, paramK, paramV, j);
  return null;
}
```

```
private V putForNullKey(V paramV)
{
  for (Entry localEntry = this.table[0]; localEntry != null; localEntry = localEntry.next) {
    if (localEntry.key == null)
    {
      Object localObject = localEntry.value;
      localEntry.value = paramV;
      localEntry.recordAccess(this);
      return (V)localObject;
    }
  }
  this.modCount += 1;
  addEntry(0, null, paramV, 0);
  return null;
}
```

addEntry(hashcode,key,value,bucket_position)

```
void addEntry(int paramInt1, K paramK, V paramV, int paramInt2)
{
  if ((this.size >= this.threshold) && (null != this.table[paramInt2]))
  {
    resize(2 * this.table.length);
    paramInt1 = null != paramK ? hash(paramK) : 0;
    paramInt2 = indexFor(paramInt1, this.table.length);
  }
  createEntry(paramInt1, paramK, paramV, paramInt2);
}
```

```
void createEntry(int paramInt1, K paramK, V paramV, int paramInt2)
{
  Entry localEntry = this.table[paramInt2];
  this.table[paramInt2] = new Entry(paramInt1, paramK, paramV, localEntry);
  this.size += 1;
}
```

Example:

Lets override default implemenation of hashcode() and equals():

You don't have to always override these methods, but lets say you want to define equality of country object based on name, then you need to override equals method and if you are overriding equals method, you should override hashcode method too. Below example will make it clear.

Lets see with the help of example.We have a class called Country

1. Country.java same as above

view plainprint?

```
1.  package org.arpit.java2blog;
2.
3.  public class Country {
4.
5.      String name;
```

```
6.      long population;
7.      public String getName() {
8.          return name;
9.      }
10.     public void setName(String name) {
11.         this.name = name;
12.     }
13.     public long getPopulation() {
14.         return population;
15.     }
16.     public void setPopulation(long population) {
17.         this.population = population;
18.     }
19.
20. }
```

This country class have two basic attributes- name and population.

Now create a class called "EqualityCheckMain.java"

view plainprint?

```
1.  package org.arpit.java2blog;
2.
3.  public class EqualityCheckMain {
4.
5.      /**
6.       * @author arpit mandliya
7.       */
8.      public static void main(String[] args) {
9.
10.         Country india1=new Country();
11.         india1.setName("India");
12.         Country india2=new Country();
13.         india2.setName("India");
14.         System.out.println("Is india1 is equal to india2:" +india1.equals(india2));
15.     }
16.
17. }
```

When you run above program, you will get following output

view plainprint?

```
1.  Is india1 is equal to india2:false
```

In above program, we have created two different objects and set their name attribute to "india".

Because both references india1 and india2 are pointing to different object, as default implementation of equals check for ==,equals method is returning false. In real life, it should have return true because no two countries can have same name.

Now lets override equals and return true if two country's name are same.

Add this method to above country class:

view plainprint?

```
@Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Country other = (Country) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
```

and now run EqualityCheckMain.java again
You will get following output:

view plainprint?

```
Is india1 is equal to india2:true
```

Now this is because overriden equals method return true if two country have same name.
One thing to remember here, signature of equals method should be same as above.

Lets put this Country objects in hashmap:

Here we are going to use Country class object as key and its capital name(string) as value in HashMap.

view plainprint?

```
package org.arpit.java2blog;

import java.util.HashMap;
```

```
4.  import java.util.Iterator;
5.
6.  public class HashMapEqualityCheckMain {
7.
8.      /**
9.  3    * @author Arpit Mandliya
10.     */
11.    public static void main(String[] args) {
12.        HashMap<Country,String> countryCapitalMap=new HashMap<Country,String>()
    ;
13.        Country india1=new Country();
14.        india1.setName("India");
15.        Country india2=new Country();
16.        india2.setName("India");
17.
18.        countryCapitalMap.put(india1, "Delhi");
19.        countryCapitalMap.put(india2, "Delhi");
20.
21.        Iterator<Country> countryCapitalIter=countryCapitalMap.keySet().iterator();
22.        while(countryCapitalIter.hasNext())
23.        {
24.            Country countryObj=countryCapitalIter.next();
25.            String capital=countryCapitalMap.get(countryObj);
26.            System.out.println("Capital of "+ countryObj.getName()+"----"+capital);
27.
28.        }
29.    }
30. }
```

When you run above program, you will see following output:

```
1.  Capital of India----Delhi
2.  Capital of India----Delhi
```

Now you must be wondering even through two objects are equal why HashMap contains two key value pair instead of one.This is because First HashMap uses hashcode to find bucket for that key object, if hashcodes are same then only it checks for equals method and because hashcode for above two country objects uses default hashcode method,Both will have different memory address hence different hashcode.

Now lets override hashcode method.Add following method to Country class

view plainprint?

```
1.  @Override
```

```
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
```

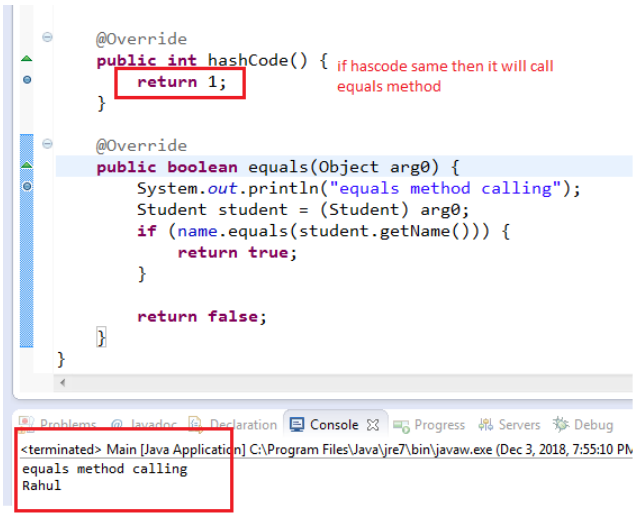
Now run HashMapEqualityCheckMain.java again
You will see following output:

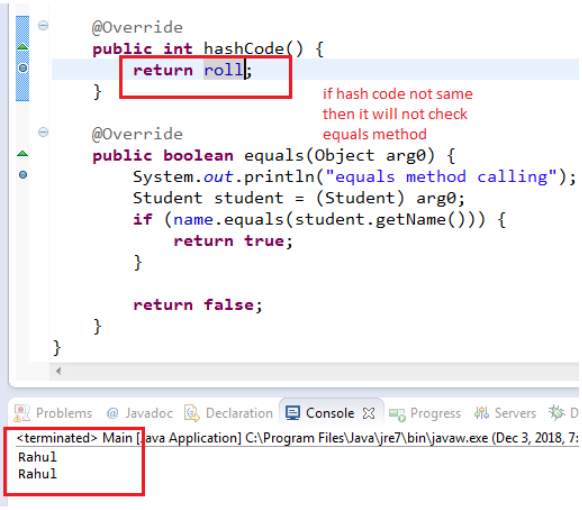
```
Capital of India----Delhi
```

So now hashcode for above two objects india1 and india2 are same, so Both will be point to same bucket,now equals method will be used to compare them which will return true. This is the reason java doc says "if you override equals() method then you must override hashCode() method"

What happened if hascode() returning same has code(means returning 1):

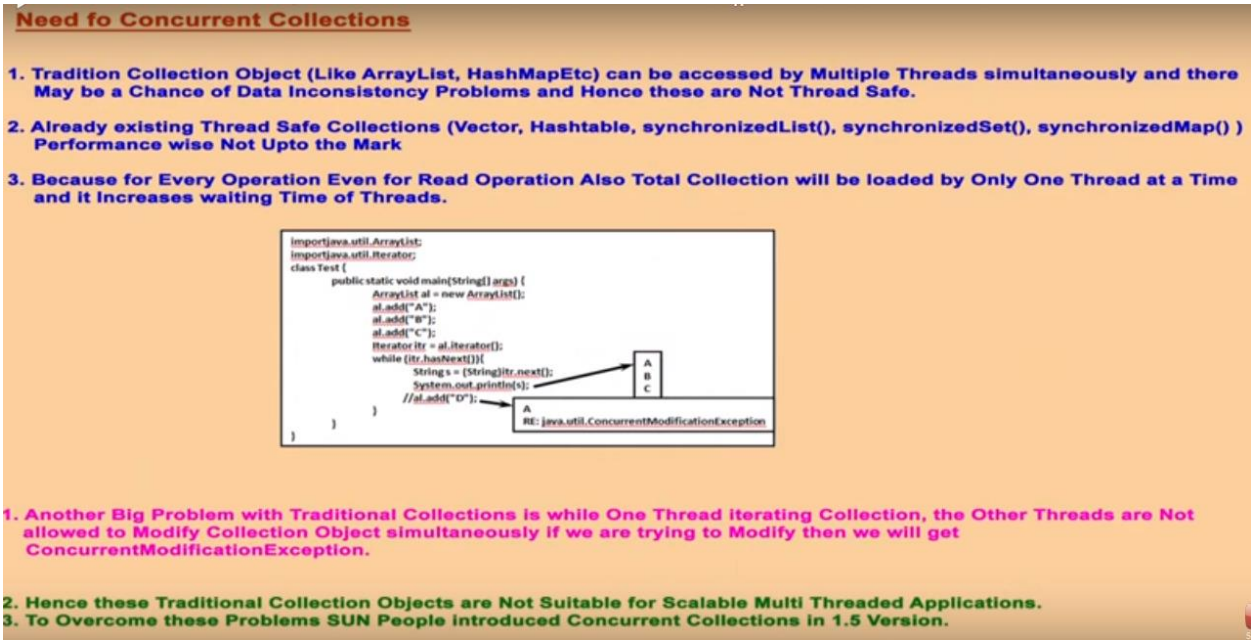
```
Map<Student, String> map=new HashMap<>();

Student s1=new Student("Rahul",1);
Student s2=new Student("Rahul",2);

map.put(s1, "Values-1");
map.put(s2, "Values-2");

Iterator<Student> iterator=map.keySet().iterator();
while(iterator.hasNext()){
    Student s=iterator.next();
    System.out.println(s.getName());
}
```



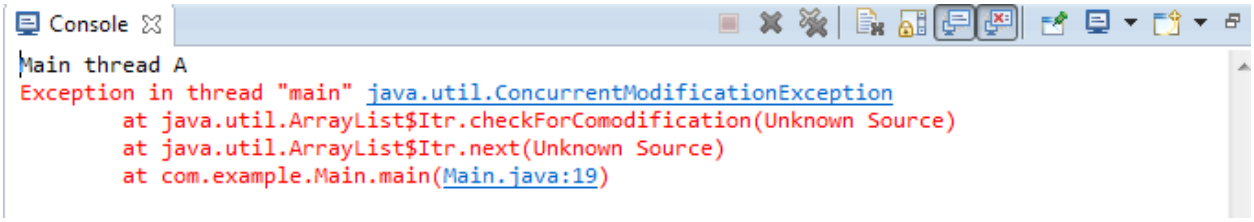


Concurrent Collections

Difference between "Traditional" & "Concurrent" Collections



```java
package com.example;
import java.util.ArrayList;
import java.util.Iterator;

public class Main extends Thread {
    static ArrayList<String> l;

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        l = new ArrayList<>();
        l.add("A");
        l.add("B");
        l.add("C");

        Main m = new Main();
        m.start();
        Iterator<String> iterator = l.iterator();
        while (iterator.hasNext()) {
            String string = (String) iterator.next();
            System.out.println("Main thread " + string);
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        super.run();
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        l.add("D");
    }
}
```



Differences Between Traditional & Concurrent Collections

1)Concurrent Collections are Always Thread Safe.

2)When compared with Traditional Thread Safe Collections Performance is More because of different Locking Mechanism.

3)While One Thread interacting Collection the Other Threads are allowed to Modify Collection in Safe Manner.

Hence Concurrent Collections Never threw ConcurrentModificationException.

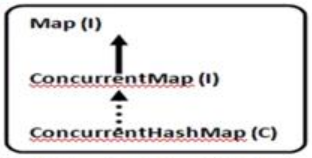
The Important Concurrent Classes are

1. ConcurrentHashMap
2. CopyOnWriteArrayList
3. CopyOnWriteArraySet

```
package java.util.concurrent;

import java.util.Map;

public abstract interface ConcurrentMap<K, V>
  extends Map<K, V>
{
  public abstract V putIfAbsent(K paramK, V paramV);

  public abstract boolean remove(Object paramObject1, Object paramObject2);

  public abstract boolean replace(K paramK, V paramV1, V paramV2);

  public abstract V replace(K paramK, V paramV);
}
```

ConcurrentMap (I):

ConcurrentMap (I):



Methods: It Defines the following 3 Specific Methods.

1) **Object putIfAbsent(Object Key, Object Value)**
To Add Entry to the Map if the specified Key is Not Already Available.

```
Object putIfAbsent(Object key, Object value)
if (!map.containsKey(key)) {
map.put(key, value);
}
else {
returnmap.get(key);
}
```

put()	putIfAbsent()
If the Key is Already Available, Old Value will be replaced with New Value and Returns Old Value.	If the Key is Already Present then Entry won't be added and Returns Old associated Value. If the Key is Not Available then Only Entry will be added.

ConcurrentMap (I):

```
importjava.util.concurrent.ConcurrentHashMap;
class Test {
        public static void main(String[] args) {
                ConcurrentHashMap m = new ConcurrentHashMap();
                m.put(101, "Durga");
                m.put(101, "Ravi");
                System.out.println(m); //{101=Ravi}
                m.putIfAbsent(101, "Siva");
                System.out.println(m); //{101=Ravi}
        }
}
```

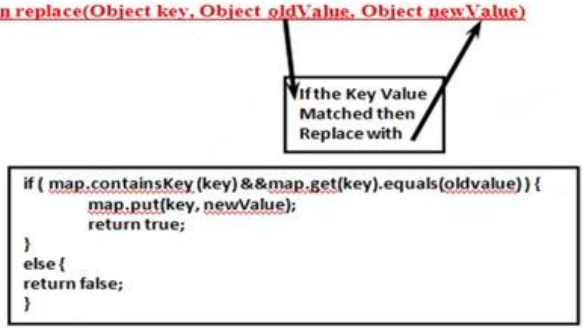
2) **boolean remove(Object key, Object value)**
Removes the Entry if the Key associated with specified Value Only.

```
if ( map.containsKey (key) &&map.get(key).equals(value) ) {
        map.remove(key);
        return true;
}
else {
return false;
}
```

```
importjava.util.concurrent.ConcurrentHashMap;
class Test {
        public static void main(String[] args) {
                ConcurrentHashMap m = new ConcurrentHashMap();
                m.put(101, "Durga");
                m.remove(101, "Ravi"); //Value Not Matched with Key So Nor Removed
                System.out.println(m); //{101=Durga}
                m.remove(101, "Durga");
                System.out.println(m); //{}
        }
```

ConcurrentMap (I):

3) boolean replace(Object key, Object oldValue, Object newValue)




```
importjava.util.concurrent.ConcurrentHashMap;
class Test {
        public static void main(String[] args) {
                ConcurrentHashMap m = new ConcurrentHashMap();
                m.put(101, "Durga");
                m.replace(101, "Ravi", "Siva");
                System.out.println(m); //{101=Durga}
                m.replace(101, "Durga", "Ravi");
                System.out.println(m); //{101=Ravi}
        }
}
```

ConcurrentHashMap

- Underlying Data Structure is Hashtable.
- ConcurrentHashMap allows Concurrent Read and Thread Safe Update Operations.
- To Perform Read Operation Thread won't require any Lock. But to Perform Update Operation Thread requires Lock but it is the Lock of Only a Particular Part of Map (Bucket Level Lock).
- Instead of Whole Map Concurrent Update achieved by Internally dividing Map into Smaller Portion which is defined by Concurrency Level.
- The Default Concurrency Level is 16.
- That is ConcurrentHashMap Allows simultaneous Read Operation and simultaneously 16 Write (Update) Operations.
- null is Not Allowed for Both Keys and Values.
- While One Thread iterating the Other Thread can Perform Update Operation and ConcurrentHashMap Never throw ConcurrentModificationException.

Constructors:

1)ConcurrentHashMap m = new ConcurrentHashMap();
Creates an Empty ConcurrentHashMap with Default Initial Capacity 16 and Default Fill Ratio
0.75and Default Concurrency Level 16.

2)ConcurrentHashMap m = new ConcurrentHashMap(intinitialCapacity);
3)ConcurrentHashMap m = new ConcurrentHashMap(intinitialCapacity, float fillRatio);
4)ConcurrentHashMap m = new ConcurrentHashMap(intinitialCapacity, float fillRatio, in-tconcurrencyLevel);
5)ConcurrentHashMap m = new ConcurrentHashMap(Map m);

ConcurrentHashMap Program-1

```
importjava.util.concurrent.ConcurrentHashMap;
class Test {
        public static void main(String[] args) {
                ConcurrentHashMap m = new ConcurrentHashMap();
                m.put(101, "A");
                m.put(102, "B");
                m.putIfAbsent(103, "C");
                m.putIfAbsent(101, "D");
                m.remove(101, "D");
                m.replace(102, "B", "E");
                System.out.println(m); //{103=C, 102=E, 101=A}
        }
}
```

ConcurrentHashMap Program-2

```
importjava.util.concurrent.ConcurrentHashMap;
importjava.util.*;
classMyThread extends Thread {
//static HashMap m = new HashMap(); // java.util.ConcurrentModificationException
staticConcurrentHashMap m = new ConcurrentHashMap();
        public void run() {
                try {
                        Thread.sleep(2000);
                }
                catch (InterruptedException e) {}
System.out.println("Child Thread updating Map");
                m.put(103, "C");
        }
        public static void main(String[] args) throws InterruptedException {
                m.put(101, "A");
                m.put(102, "B");
                MyThread t = new MyThread();
                t.start();
                Set s = m.keySet();
                Iterator itr = s.iterator();
                while (itr.hasNext())     {
                        Integer I1 = (Integer) itr.next();
                SOP("Main Thread iterating and Current Entry is:"+I1+"....."+m.get(I1));
                        Thread.sleep(3000);
                }
                System.out.println(m);
        }
}
```

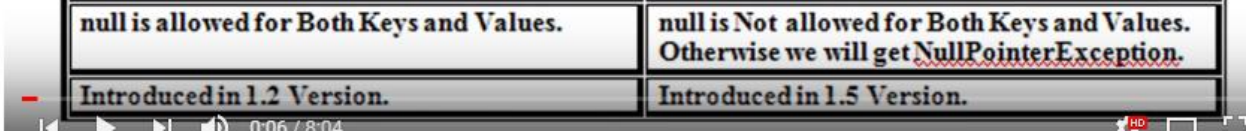
```
Main Thread iterating and Current Entry is:102.....B
Child Thread updating Map
Main Thread iterating and Current Entry is:101.....A
{103=C, 102=B, 101=A}
```

Update and we won't get any *ConcurrentModificationException*.

- If we Replace ConcurrentHashMap with HashMap then we will get *ConcurrentModificationException*.

Difference between HashMap & ConcurrentHashMap

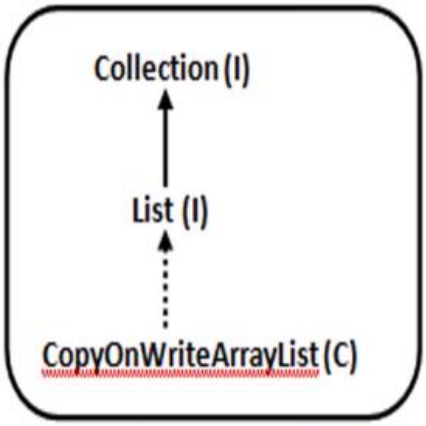
HashMap	ConcurrentHashMap
It is Not Thread Safe.	It is Thread Safe.
Relatively Performance is High because Threads are Not required to wait to Operate on HashMap.	Relatively Performance is Low because Some Times Threads are required to wait to Operate on ConcurrentHashMap.
While One Thread iterating HashMap the Other Threads are Not allowed to Modify Map Objects Otherwise we will get Runtime Exception Saying ConcurrentModificationException.	While One Thread iterating ConcurrentHashMap the Other Threads are allowed to Modify Map Objects in Safe Manner and it won't throw ConcurrentModificationException.
Iterator of HashMap is Fail-Fast and it throws ConcurrentModificationException.	Iterator of ConcurrentHashMap is Fail-Safe and it won't throwsConcurrentModificationException.
null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values. Otherwise we will get NullPointerException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.



Difference between ConcurrentHashMap, synchronizedMap() and Hashtable

ConcurrentHashMap	synchronizedMap()	Hashtable
We will get Thread Safety without locking Total Map Object Just with Bucket Level Lock.	We will get Thread Safety by locking Whole Map Object.	We will get Thread Safety by locking Whole Map Object.
At a Time Multiple Threads are allowed to Operate on Map Object in Safe Manner.	At a Time Only One Thread is allowed to Perform any Operation on Map Object.	At a Time Only One Thread is allowed to Operate on Map Object.
Read Operation can be performed without Lock but write Operation can be performed with Bucket Level Lock.	Every Read and Write Operations require Total Map Object Lock.	Every Read and Write Operations require Total Map Object Lock.
While One Thread iterating Map Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.	While One Thread iterating Map Object, the Other Threads are Not allowed to Modify Map. Otherwise we will get ConcurrentModificationException	While One Thread iterating Map Object, the Other Threads are Not allowed to Modify Map. Otherwise we will get ConcurrentModificationException
Iterator of ConcurrentHashMap is Fail-Safe and won't raise ConcurrentModificationException.	Iterator of synchronizedMap is Fail-Fast and it will raise ConcurrentModificationException.	Iterator of synchronizedMap is Fail-Fast and it will raise ConcurrentModificationException.
null is Not allowed for Both Keys and Values.	null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArrayList (C):



- It is a Thread Safe Version of ArrayList as the Name indicates CopyOnWriteArrayList Creates a Cloned Copy of Underlying ArrayList for Every Update Operation at Certain Point Both will Synchronized Automatically Which is taken Care by JVM Internally.

1

- As Update Operation will be performed on cloned Copy there is No Effect for the Threads which performs Read Operation.
- It is Costly to Use because for every Update Operation a cloned Copy will be Created. Hence CopyOnWriteArrayList is the Best Choice if Several Read Operations and Less Number of Write Operations are required to Perform.
- Insertion Order is Preserved.
- Duplicate Objects are allowed.
- Heterogeneous Objects are allowed.
- null Insertion is Possible.
- It implements Serializable, Clonable and RandomAccess Interfaces.
- While One Thread iterating CopyOnWriteArrayList, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException. That is iterator is Fail Safe.
- Iterator of ArrayList can Perform Remove Operation but Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException Saying UnsupportedOperationException.

2

Constructors:

1) CopyOnWriteArrayList l = new CopyOnWriteArrayList();
2) CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);
3) CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);

Methods:

1. booleanaddIfAbsent(Object o): The Element will be Added if and Only if List doesn't contain this Element.

```
CopyOnWriteArrayList l = new CopyOnWriteArrayList();
l.add("A");
l.add("A");
l.addIfAbsent("B");
l.addIfAbsent("B");
System.out.println(l); //[A, A, B]
```

2. intaddAllAbsent(Collection c): The Elements of Collection will be Added to the List if Elements are Absent and Returns Number of Elements Added.

```
ArrayList l = new ArrayList();
l.add("A");
l.add("B");

CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();
l1.add("A");
l1.add("C");
System.out.println(l1); //[A, C]
l1.addAll(l);
System.out.println(l1); //[A, C, A, B]

ArrayList l2 = new ArrayList();
l2.add("A");
l2.add("D");
l1.addAllAbsent(l2);
System.out.println(l1); //[A, C, A, B, D]
```

```
importjava.util.concurrent.CopyOnWriteArrayList;
importjava.util.ArrayList;
class Test {
        public static void main(String[] args) {
ArrayList l = new ArrayList();
                l.add("A");
                l.add("B");

                CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();
l1.addIfAbsent("A");
l1.addIfAbsent("C");
                l1.addAll(l);

ArrayList l2 = new ArrayList();
                l2.add("A");
        l2.add("E");
                l1.addAllAbsent(l2);

                System.out.println(l1); //[A, C, A, B, E]
}
}
```

Note: In above example l1.addAll(l)----------is the normal arraylist method that's why it will not check already added elements. Means it will directly addAll elements of the "l" into "l1" object.

```
importjava.util.concurrent.CopyOnWriteArrayList;
importjava.util.*;
classMyThread extends Thread {
staticCopyOnWriteArrayList l = new CopyOnWriteArrayList();
        public void run() {
                try {  Thread.sleep(2000);  }
                catch (InterruptedException e) {}
System.out.println("Child Thread Updating List");
                l.add("C");
         }
        public static void main(String[] args) throws InterruptedException {
        l.add("A");
        l.add("B");
                MyThread t = new MyThread();
                t.start();
                Iterator itr = l.iterator();
                while (itr.hasNext()) {
                        String s1 = (String)itr.next();
                        System.out.println("Main Thread Iterating List And Current Object is: " +s1);
                        Thread.sleep(3000);
                }
                System.out.println(l);
}
}
```

```
Main Thread Iterating List And Current Object is: A
Child Thread Updating List
Main Thread Iterating List And Current Object is: B
[A, B, C]
```

- **In the Above Example while Main Thread iterating List Child Thread is allowed to Modify and we won't get any ConcurrentModificationException.**

•If we Replace CopyOnWriteArrayList with ArrayList then we will get ConcurrentModificationException.

•Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperationException.

```
importjava.util.concurrent.CopyOnWriteArrayList;
importjava.util.Iterator;
class Test {
public static void main(String[] args){
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        l.add("D");
        System.out.println(l); //[A, B, C, D]
        Iterator itr = l.iterator();
        while (itr.hasNext()) {
                String s = (String)itr.next();
                if (s.equals("D"))
                        itr.remove();
        }
        System.out.println(l); //RE: java.lang.UnsupportedOperationException
}
}
```

- **If we Replace CopyOnWriteArrayList with ArrayList we won't get any UnsupportedOperationException.**
- **In this Case the Output is**
  - **[A, B, C, D]**
  - **[A, B, C]**

```
importjava.util.concurrent.CopyOnWriteArrayList;
importjava.util.Iterator;
class Test {
public static void main(String[] args) {
        CopyOnWriteArrayList l = new CopyOnWriteArrayList();
        l.add("A");
        l.add("B");
        l.add("C");
        Iterator itr = l.iterator();
        l.add("D");
        while (itr.hasNext()) {
                String s = (String)itr.next();
                System.out.println(s);
        }
}
}
```

```
A
B
C
```

Reason:

- **Every Update Operation will be performed on Separate Copy Hence After getting iterator if we are trying to Perform any Modification to the List it won't be reflected to the iterator.**
- **In the Above Program if we ReplaceCopyOnWriteArrayList with ArrayList then we will get RuntimeException: java.util.ConcurrentModificationException.**

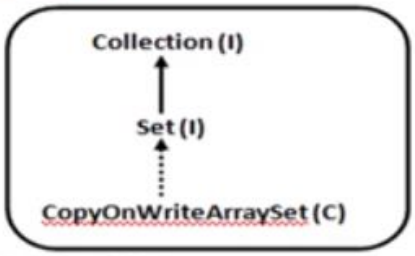
Differences between ArrayList and CopyOnWriteArrayList

ArrayList	CopyOnWriteArrayList
It is Not Thread Safe.	It is Not Thread Safe because Every Update Operation will be performed on Separate cloned Coy.
While One Thread iterating List Object, the Other Threads are Not allowed to Modify List Otherwise we will get ConcurrentModificationException.	While One Thread iterating List Object, the Other Threads are allowed to Modify List in Safe Manner and we won't get ConcurrentModificationException.
Iterator is Fail-Fsat.	Iterator is Fail-Safe.
Iterator of ArrayList can Perform Remove Operation.	Iterator of CopyOnWriteArrayList can't Perform Remove Operation Otherwise we will get RuntimeException: UnsupportedOperationException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.

Differences between CopyOnWriteArrayList, synchronizedList() and vector()

CopyOnWriteArrayList	synchronizedList()	vector()
We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.	We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.	We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.
At a Time Multiple Threads are allowed to Access/ Operate on CopyOnWriteArrayList.	At a Time Only One Thread is allowed to Perform any Operation on List Object.	At a Time Only One Thread is allowed to Operate on Vector Object.
While One Thread iterating List Object, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.	While One Thread iterating , the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException	While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException
Iterator is Fail-Safe and won't raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.
Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.	Iterator canPerform Remove Operation.	Iterator can Perform Remove Operation.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArraySet :



- It is a Thread Safe Version of Set.
- Internally Implement by CopyOnWriteArrayList.
- Insertion Order is Preserved.
- Duplicate Objects are Notallowed.
- Multiple Threads can Able to Perform Read Operation simultaneously but for Every Update Operation a Separate cloned Copy will be Created.
- As for Every Update Operation a Separate cloned Copy will be Created which is Costly Hence if Multiple Update Operation are required then it is Not recommended to Use CopyOnWriteArraySet.
- While One Thread iterating Set the Other Threads are allowed to Modify Set and we won't get ConcurrentModificationException.
- Iterator of CopyOnWriteArraySet can PerformOnly Read Operation and won't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperatonException.

Constructors:

1) CopyOnWriteArraySets = new CopyOnWriteArraySet();
Creates an Empty CopyOnWriteArraySet Object.

2) CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);
Creates CopyOnWriteArraySet Object which is Equivalent to given Collection Object.

Methods:Whatever Methods Present in Collection and Set Interfaces are the Only Methods Applicable for CopyOnWriteArraySet and there are No Special Methods.

```
importjava.util.concurrent.CopyOnWriteArraySet;
class Test {
public static void main(String[] args) {
CopyOnWriteArraySet s = new CopyOnWriteArraySet();
                    s.add("A");
                    s.add("B");
                    s.add("C");
                    s.add("A");
                    s.add(null);
                    s.add(10);
                    s.add("D");
```

Differences between CopyOnWriteArraySet() and synchronizedSet()

CopyOnWriteArraySet()	synchronizedSet()
It is Thread Safe because Every Update Operation will be performed on Separate Cloned Copy.	It is Thread Safe because at a Time Only One Thread can Perform Operation.
While One Thread iterating Set, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify Seta Otherwise we will get ConcurrentModificationException.
Iterator is Fail Safe.	Iterator is Fail Fast.
Iterator can Perform Only Read Operation and can't Perform Remove Operation Otherwise we will get RuntimeException Saying UnsupportedOperationException.	Iterator can Perform Both Read and Remove Operations.
Introduced in 1.5 Version.	Introduced in 1.2 Version.

Fail Fast Iterator

Multithreading:

Multithreading is a process of executing two or more parts of a program simultaneously to maximum utilization of the CPU. Each part of such a program is called a **thread**.

Synchronization:

In Java, **synchronization** is the mechanism that controls the access of multiple threads to shared resources. Without it, you might encounter "**Race Conditions**," where two threads modify the same data simultaneously, leading to unpredictable results.

Think of it like a **single-person restroom**: the lock on the door ensures that only one person can enter at a time. In Java, this "lock" is known as a **Monitor**.

Race Condition:

Race Condition ek aisi situation hai jaha do ya do se zyada threads ek hi shared data par kaam karne ki koshish karte hain, aur final result is baat par depend karta hai ki kaunsa thread pehle execute hua.

Isse program mein unpredictable bugs aate hain kyunki result hamesha "race" (kaun jeeta) par nirbhar karta hai.

Ek Real-Life Example

Maano ek Bank Account mein ₹100 hain. Do log (Threads) ek saath ₹50 nikalne ki koshish karte hain:

1. **Thread A** check karta hai: Balance ₹100 hai? Haan.
2. **Thread B** bhi check karta hai: Balance ₹100 hai? Haan.
3. **Thread A** ₹50 nikal leta hai aur balance ₹50 update kar deta hai.
4. **Thread B** (jisne pehle hi ₹100 dekha tha) ₹50 nikalta hai aur balance ko phir se ₹50 update kar deta hai.

Problem: Total ₹100 nikal gaye, lekin balance abhi bhi ₹50 dikha raha hai! Yeh data inconsistency hi Race Condition hai.

Java mein Synchronization kyun zaroori hai?

Synchronization ka maqsad race conditions ko rokna hai. Java mein hum `synchronized` keyword ka use karte hain taaki ek waqt mein sirf **ek hi thread** critical section (shared data) ko access kar sake.

Method Level Synchronization:

A. Synchronized Methods

By adding the `synchronized` keyword to a method header, you lock the **entire method** for the current object (`this`).

Java

```java
public synchronized void deposit(int amount) {
    balance += amount;
}
```

Block Level Synchronization:

B. Synchronized Blocks

If you only need to protect a few lines of code rather than an entire method, synchronized blocks are more efficient. They allow you to specify exactly which object to lock on.

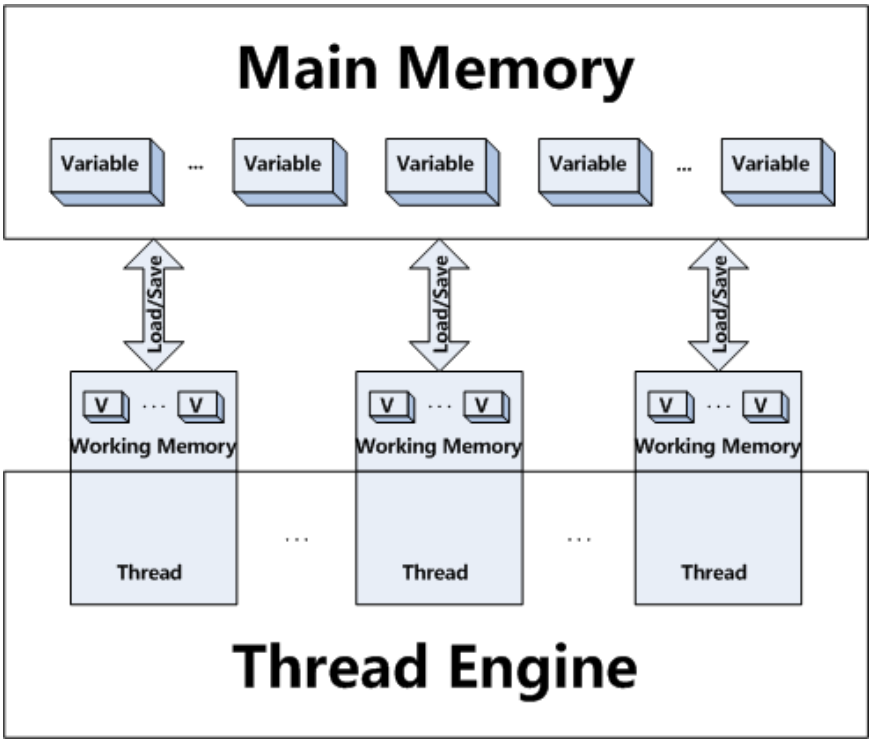
Java

```
public void updateData() {
    // Non-critical code here (accessible by all threads)

    synchronized(this) {
        // Critical section (only one thread at a time)
        this.count++;
    }
}
```

Volatile keyword:

- Synchronization in Java is possible by using Java keywords **synchronized** and **volatile**.
- In Java, we can not have **synchronized** variable. Using synchronized keyword with a variable is illegal and will result in compilation error. Instead of using the synchronized variable in Java, you can use the java **volatile** variable, which will instruct JVM threads to read the value of volatile variable from main memory and don't cache it locally.
- If a variable is not shared between multiple threads then there is no need to use volatile keyword.
- Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.
- Volatile Variables are light-weight synchronization. When visibility of latest data among all threads is requirement and atomicity can be compromised , in such situations Volatile Variables must be preferred. Read on volatile variables always return most recent write done by any thread since they are neither cached in registers nor in caches where other processors can not see. Volatile is Lock-Free.

Note: In volatile first priority is to always write after read operation.



Differences Between Synchronized and Volatile:

1. The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.

2. Synchronized obtains and releases the lock on monitor's Java volatile keyword doesn't require that.

3. Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the volatile keyword in Java.

4. Synchronized method affects performance more than a volatile keyword in Java.

5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and `"main"` memory while synchronized synchronizes the value of all variable between thread memory and `"main"` memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.

6. You can not synchronize on the `null` object but your volatile variable in Java could be null.

7. From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

Example Singleton Class in case of Multithreading:

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton() {
    }
    public static Singleton getInstance() {
        if (singleton == null) {
            // To make thread safe
            synchronized (Singleton.class) {
                // check again as multiple threads can reach above step
                if (singleton == null)
                    singleton = new Singleton();
            }
        }
        return singleton;
    }
}
```

```
public class Test {

    public static void main(String[] args) {
        ExecutorService executorService = null;
        try {
            executorService = Executors.newFixedThreadPool(5);

            Runnable t1 = new Runnable() {

                @Override
                public void run() {
                    for (int i = 1; i <=100; i++) {
                        Singleton singleton = Singleton.getInstance();
                        System.out.println(singleton.hashCode());
                    }
                }
            };


            Runnable t2 = new Runnable() {

                @Override
                public void run() {
                    for (int i = 1; i <=100; i++) {
                        Singleton singleton = Singleton.getInstance();
                        System.out.println(singleton.hashCode());
                    }
                }
            };
```

```
            executorService.submit(t1);
            executorService.submit(t2);
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            if(executorService != null)
                executorService.shutdown();
        }
    }

}
```

The hashcode will be the same because instance create only once.

Note: Volatile gives a visibility but not the atomicity.

Atomic Variable:

Atomic variables are classes provided in the java.util.concurrent.atomic package that allow you to perform thread-safe operations on single variables without using locks (like synchronized or ReentrantLock).

Example-#1

Problem:

Problem: Normal Variable ke saath (Race Condition)

Sochiye hamare paas ek variable hai `int count = 0;`. Do threads (`Thread 1` aur `Thread 2`) isko increment (`count++`) karna chahti hain.

Asal mein `count++` ek step nahi, balki **3 steps** hota hai:

1. **Read:** Memory se value uthao (maan lo `0`).
2. **Modify:** Usme `+1` karo (ab value `1` ho gayi).
3. **Write:** Wapas memory mein save karo.

Solution:

Solution: AtomicInteger ke saath (CAS Mechanism)

Ab wahi kaam `AtomicInteger` se karte hain. Ye **Compare-And-Swap (CAS)** ka use karta hai. Iska logic hota hai: *"Value tabhi update karo agar purani value wahi hai jo maine read ki thi."*

Example with Atomic:

1. **Thread 1** ne read kiya: `0` .
2. **Thread 2** ne read kiya: `0` .
3. **Thread 1** ne update karne ki koshish ki. Usne pucha: *"Kya memory mein abhi bhi 0 hai?"*
   - System ne kaha: *"Haan"*.
   - **Thread 1** ne use `1` kar diya. (Memory = 1).
4. **Thread 2** ne update karne ki koshish ki. Usne pucha: *"Kya memory mein abhi bhi 0 hai?"* (Kyuki usne 0 read kiya tha).
   - System ne kaha: **"Nahi, ab wahan 1 ho chuka hai!"**
5. **Thread 2** haar nahi maanta. Wo phir se nayi value (`1`) read karta hai aur dobara koshish karta hai. Is baar wo successfully use `2` kar deta hai.

AtomicInteger

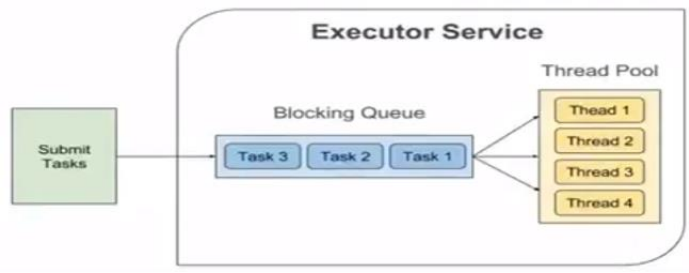
```java
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
    // Atomic variable
    AtomicInteger count = new AtomicInteger(0);

    public void doWork() {
        // Thread 1
        Thread t1 = new Thread(() -> {
            for(int i=0; i<1000; i++) count.incrementAndGet();
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for(int i=0; i<1000; i++) count.incrementAndGet();
        });

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

        // Wait for both to finish...
        // Result will ALWAYS be 2000
    }
}
```

Thread Pool:

Most of the executor implementations use thread pools to execute tasks, A thread pool is nothing but a bunch of worker threads that exist separately from the runnable or Callable tasks and is managed by the executor.

Creating a thread is an expensive operation and it should be minimized. Having worker threads minimizes the overhead due to thread creation because executor service has to create the thread pool only once and then it can reuse the threads for executing any tasks.

Tasks are submitted to a thread pool via an internal queue called the Blocking queue. If there are more tasks then for waiting until any thread becomes available. If the blocking queue is full than new tasks are rejected



java.util.concurrent:

The java.util.concurrent (J.U.C) package is organized into several distinct sub-hierarchies. Rather than one single tree, it is a collection of frameworks designed to handle specific threading problems like task scheduling, synchronization, and thread-safe data storage.

It was introduced in JDK 1.5 release as an alternative of synchronized keyword. As I told you Lock is an interface, so we cannot use it directly, instead we need to use its implementation class

Future and Callable Concurrency:

java.util.concurrent.Callable object can return the computed result done by a thread in contrast to runnable interface which can only run the thread. The Callable object returns Future object which provides methods to monitor the progress of a task being executed by a thread. Future object can be used to check the status of a Callable and then retrieve the result from the Callable once the thread is done. It also provides timeout functionality.

Callable vs Runnable interface in Java

As I explained major differences between a Callable and Runnable interface in the last section. Sometimes this question is also asked as the difference between call() and run() method in Java. All the points discussed here is equally related to that question as well. Let's see them in point format for better understanding :

1) The Runnable interface is older than Callable, there from JDK 1.0, while Callable is added on Java 5.0.

2) Runnable interface has run() method to define task while Callable interface uses call() method for task definition.

3) run() method does not return any value, it's return type is void while call method returns value. The Callable interface is a generic parameterized interface and Type of value is provided when an instance of Callable implementation is created.

4) Another difference on run and call method is that run method can not throw checked exception while call method can throw checked exception in Java.

Here is a nice summary of all the **differences between Callable and Runnable in Java**:

```
public Object call() throws Exception;
```

Both callable and runnable interface are the Functional Interface(annotation of the java 7(functionalinteface)).

Runnable interfaces belong to java.lang package.

1. In this runnable run method don't throws any exception.
2. Does not return any value.

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

Callable interface belong to java.util.concurrent. package.

1. In this callable call() method throws Exeption.
2. It return future value means return object.
3. Return type "v" represent Future Interace.

```
public interface Future<V> {

    /**
     * Attempts to cancel execution of this t
```

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Example:

Output: Future Result : My Callable Interface Task

1. The Executor Framework (Task Execution):

Managing thread pools and asynchronous tasks. It separates task submission from task execution.

Executors, A framework for creating and managing threads. Executors framework helps you with –

- **Thread Creation**: It provides various methods for creating threads, more specifically a pool of threads, that your application can use to run tasks concurrently.
- **Thread Management**: It manages the life cycle of the threads in the thread pool. You don't need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution.
- **Task submission and execution**: Executors framework provides methods for submitting tasks for execution in the thread pool, and also gives you the power to decide when the tasks will be executed. For example, You can submit a task to be executed now or schedule them to be executed later or make them execute periodically.

Java Concurrency API defines the following three executor interfaces that covers everything that is needed for creating and managing threads –

- **Executor** – A simple interface that contains a method called execute(Runnable command) to launch a task specified by a Runnable object.
- **ExecutorService** – A sub-interface of Executor that adds functionality to manage the lifecycle of the tasks. It also provides a submit() method whose overloaded versions can accept a Runnable as well as a Callable object. Callable objects are similar to Runnable except that the task specified by a Callable object can also return a value.
- **ScheduledExecutorService** – A sub-interface of ExecutorService. It adds functionality to schedule the execution of the tasks.

ExecutorService provides two methods for shutting down an executor –

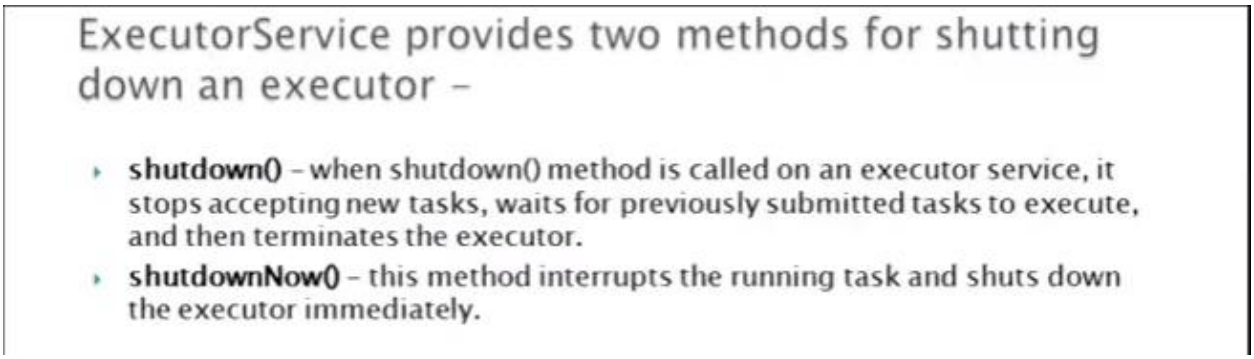
- **shutdown()** – when shutdown() method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
- **shutdownNow()** – this method interrupts the running task and shuts down the executor immediately.

2.Executor Interface:

```
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

3.ExecutorService Interface:

```
public abstract interface ExecutorService
  extends Executor
{
  public abstract void shutdown();

  public abstract List<Runnable> shutdownNow();

  public abstract boolean isShutdown();

  public abstract boolean isTerminated();

  public abstract boolean awaitTermination(long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException;

  public abstract <T> Future<T> submit(Callable<T> paramCallable);

  public abstract <T> Future<T> submit(Runnable paramRunnable, T paramT);

  public abstract Future<?> submit(Runnable paramRunnable);

  public abstract <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> paramCollection)
    throws InterruptedException;

  public abstract <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> paramCollection, long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException;

  public abstract <T> T invokeAny(Collection<? extends Callable<T>> paramCollection)
    throws InterruptedException, ExecutionException;

  public abstract <T> T invokeAny(Collection<? extends Callable<T>> paramCollection, long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException, ExecutionException, TimeoutException;
}
```



Here are examples to creates ExecutorService and ScheduledExecutorService instances:

```
// Creates a single thread ExecutorService
ExecutorService singleExecutorService = Executors.newSingleThreadExecutor();

// Creates a single thread ScheduledExecutorService
ScheduledExecutorService singleScheduledExecutorService = Executors.newSingleThreadScheduledExecutor();

// Creates an ExecutorService that use a pool of 10 threads
ExecutorService fixedExecutorService = Executors.newFixedThreadPool(10);

// Creates an ExecutorService that use a pool that creates threads on demand
// And that kill them after 60 seconds if they are not used
ExecutorService onDemandExecutorService = Executors.newCachedThreadPool();

// Creates a ScheduledExecutorService that use a pool of 5 threads
ScheduledExecutorService fixedScheduledExecutorService = Executors.newScheduledThreadPool(5);
```

4.ScheduledExecutorService Interface:

```
public abstract interface ScheduledExecutorService
  extends ExecutorService
{
  public abstract ScheduledFuture<?> schedule(Runnable paramRunnable, long paramLong, TimeUnit paramTimeUnit);

  public abstract <V> ScheduledFuture<V> schedule(Callable<V> paramCallable, long paramLong, TimeUnit paramTimeUnit);

  public abstract ScheduledFuture<?> scheduleAtFixedRate(Runnable paramRunnable, long paramLong1, long paramLong2, TimeUnit paramTimeUnit);

  public abstract ScheduledFuture<?> scheduleWithFixedDelay(Runnable paramRunnable, long paramLong1, long paramLong2, TimeUnit paramTimeUnit);
}
```

Executors Utility Class:

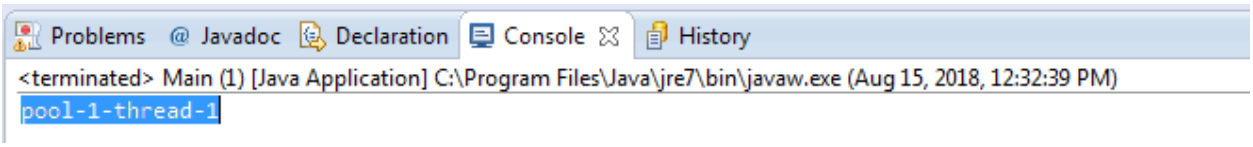
Apart from the above three interfaces, the API also provides an Executors class that contains factory methods for creating different kinds of executor services.

Example #1.

Store all threads into the thread pool.

Here we are using single thread only. So it will create single thread only into the thread pool.

```java
package com.executorsevice.framwork;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
public static void main(String[] args) {
    ExecutorService executorService= Executors.newSingleThreadExecutor();
     Runnable task1=new Runnable() {

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName());
        }
    };
    executorService.submit(task1);
    executorService.shutdown();
}
}
```



Example #2:

Java ExecutorService with a pool of threads for executing multiple tasks:

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
public static void main(String[] args) {
    ExecutorService executorService= Executors.newFixedThreadPool(3);
     Runnable task1=new Runnable() {

        @Override
        public void run() {
            System.out.println("First Task Started : "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("First Task Ended : "+Thread.currentThread().getName());
        }
    };
    Runnable task2=new Runnable() {

        @Override
        public void run() {
            System.out.println("Second Task Started : "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Second Task Ended : "+Thread.currentThread().getName());
        }
    };
```

```
    Runnable task3=new Runnable() {

        @Override
        public void run() {
            System.out.println("Third Task Started : "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Third Task Ended : "+Thread.currentThread().getName());
        }
    };
    executorService.submit(task1);
    executorService.submit(task2);
    executorService.submit(task3);
    executorService.shutdown();
}
}
```

Problems @ Javadoc Declaration Console History

<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Aug 15, 2018, 12:42:54 PM)

```
First Task Started : pool-1-thread-1
Second Task Started : pool-1-thread-2
Third Task Started : pool-1-thread-3
First Task Ended : pool-1-thread-1
Third Task Ended : pool-1-thread-3
Second Task Ended : pool-1-thread-2
```

Java Concurrency - Lock Interface:

A java.util.concurrent.locks.Lock interface is used to as a thread synchronization mechanism similar to synchronized blocks. New Locking mechanism is more flexible and provides more options than a synchronized block. Main differences between a Lock and a synchronized block are following –

- **Guarantee of sequence** – Synchronized block does not provide any guarantee of sequence in which waiting thread will be given access. Lock interface handles it.
- **No timeout** – Synchronized block has no option of timeout if lock is not granted. Lock interface provides such option.
- **Single method** – Synchronized block must be fully contained within a single method whereas a lock interface's methods lock() and unlock() can be called in different methods.

1. Key Methods of the `Lock` Interface

Unlike `synchronized`, which is a language-level keyword, `Lock` is an interface that requires explicit calls to acquire and release locks.

Method	Description
`void lock()`	Acquires the lock. Blocks the thread indefinitely until the lock is available.
`boolean tryLock()`	**Non-blocking.** Attempts to acquire the lock and returns `true` immediately if successful, `false` otherwise.
`boolean tryLock(long time, TimeUnit unit)`	Attempts to acquire the lock within a specific timeframe before giving up.
`void lockInterruptibly()`	Acquires the lock unless the thread is interrupted, allowing it to respond to `Thread.interrupt()`.
`void unlock()`	Releases the lock. **Must** be called in a `finally` block to avoid deadlocks.
`Condition newCondition()`	Returns a `Condition` instance for advanced inter-thread communication (replaces `wait/notify`).

ReentrantLock

ReentrantLock is the most common implementation. "Reentrant" means a thread can re-acquire the same lock it already holds without deadlocking itself (it simply increments a "hold count").

A **ReentrantLock** is a concrete implementation of the Lock interface in the java.util.concurrent.locks package. It is called "reentrant" because it allows a thread to acquire the same lock multiple times without deadlocking itself.

```
Lock lock = new ReentrantLock();

public void performTask() {
    lock.lock(); // Explicitly acquire
    try {
        // Critical Section (shared resource access)
    } finally {
        lock.unlock(); // Always release in finally to ensure safety
    }
}
```

1. How Reentrancy Works

Every time a thread acquires the lock, a **hold count** is incremented by 1. Every time the thread releases the lock, the count is decremented. The lock is only truly released for other threads when the hold count reaches zero.

Example Scenario: If Method A (synchronized on a lock) calls Method B (also synchronized on the same lock), a reentrant lock allows the thread to enter Method B because it already owns the lock.

```
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock(); // Block until acquired
        try {
            count++;
        } finally {
            lock.unlock(); // Ensure release even if exception occurs
        }
    }

    public int getCount() {
        return count;
    }
}
```

Blocking Queues (Data Transfer):

Before BlockingQueue, you had to manually use synchronized, wait(), and notifyAll() to manage buffers, which is extremely error-prone and leads to bugs like **Lost Wakeups** or **Spurious Wakeups**. BlockingQueue encapsulates all that complexity into a single, thread-safe class.

Blocking Queues are the primary tool for solving the **Producer-Consumer** problem. They act as a thread-safe "buffer" that handles the coordination of data transfer between threads automatically.

What makes them "Blocking" is their behavior when the queue is full or empty:

- **When Empty:** A thread trying to **take**() an element will block (wait) until a producer adds something.
- **When Full:** A thread trying to **put**() an element will block (wait) until a consumer removes something (for bounded queues).

2. Blocking Queues (Data Transfer)

These queues are used for the **Producer-Consumer pattern**. They automatically "block" a thread if it tries to take from an empty queue or put into a full one.

- `BlockingQueue` **(Interface):** The core interface for thread-safe queues.
  - `ArrayBlockingQueue` : Bounded (fixed size), backed by an array.
  - `LinkedBlockingQueue` : Optionally bounded, backed by linked nodes (higher throughput).
  - `PriorityBlockingQueue` : Unbounded, elements are ordered by priority.
  - `SynchronousQueue` : A queue where each insert must wait for a corresponding remove (direct hand-off).

Synchronizers:

Synchronizers are high-level coordination utilities that facilitate common interaction patterns between threads. They allow threads to wait for each other, communicate state changes, or manage access to shared resources without the complexity of low-level wait() and notify() calls.

Here is the breakdown of the primary synchronizers found in java.util.concurrent.

1.CountDownLatch:

Real-life example aisa samajhiye: **Ek race tabhi start hogi jab saare (N) runners starting line par pahunch jayenge.**

CountDownLatch in Java is a kind of synchronizer which allows one Thread to wait for one or more Threads before starts processing. This is very crucial requirement and often needed in server side core Java application and having this functionality built-in as CountDownLatch greatly simplifies the development. CountDownLatch in Java is introduced on Java 5.

When should we use CountDownLatch in Java :

Use CountDownLatch when one of Thread like main thread, require to wait for one or more thread to complete, before its start doing processing. Classical example of using CountDownLatch in Java is any server side core Java application which uses services architecture, where multiple services is provided by multiple threads and application cannot start processing until all services have started successfully as shown in our CountDownLatch example.**countDownLatch in Java – Things to remember:**

Few points about Java CountDownLatch which is worth remembering:

1. You can not reuse CountDownLatch once count is reaches to zero, this is the main difference between CountDownLatch and CyclicBarrier, which is frequently asked in core Java interviews and multi-threading interviews.

2. Main Thread wait on Latch by calling CountDownLatch.await() method while other thread calls CountDownLatch.countDown() to inform that they have completed.

Example: IT and BA task.

```java
package com.infotech.client;

import java.util.concurrent.CountDownLatch;

public class AssignTaskManagerTest {
    public static void main(String[] args) throws InterruptedException {
        //Created CountDownLatch for 2 threads
        CountDownLatch countDownLatch = new CountDownLatch(2);

        //Created and started two threads
        DevTeam teamDevA = new DevTeam(countDownLatch, "dev-A");
        DevTeam teamDevB = new DevTeam(countDownLatch, "dev-B");

        teamDevA.start();
        teamDevB.start();

        //When two threads(dev-A and dev-B)completed tasks are returned
        countDownLatch.await();

        //Now execution of thread(QA team) started..
        QATeam qaTeam = new QATeam("QA team");
        qaTeam.start();
    }
}
```

```java
package com.infotech.tasks;

public class QATeam extends Thread {

    public QATeam(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("Task assigned to "+Thread.currentThread().getName());
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("Task finished by "+Thread.currentThread().getName());
    }
}
```

2.CyclicBarrier:

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc. CyclicBarrier is similar to CountDownLatch and allows multiple threads to wait for each other (barrier) before proceeding. The difference between CountDownLatch . CyclicBarrier is a natural requirement for a concurrent program because it can be used to perform final part of the task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with a number of parties to wait and threads wait for each other by calling CyclicBarrier.await() method which is a blocking method in Java and blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on the barrier. await() is a blocking call but can be timed out or Interrupted by other thread.

Difference between CountDownLatch and CyclicBarrier in Java:

If you look at CyclicBarrier it also the does the same thing but there is different you can not reuse CountDownLatch once the count reaches zero while you can reuse CyclicBarrier by calling reset() method which resets Barrier to its initial State. What it implies that CountDownLatch is a good for one-time events like application start-up time and CyclicBarrier can be used to in case of the recurrent event e.g. concurrently calculating a solution of the big problem etc.

Important point of CyclicBarrier in Java:

1. CyclicBarrier can perform a completion task once all thread reaches to the barrier, This can be provided while creating CyclicBarrier.

2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.

3. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.

4. If another thread interrupts the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:

java.util.concurrent.BrokenBarrierException

at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)

at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)

5.CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

Example:

```
public class PassengerThread extends Thread {
    private int duration;
    private CyclicBarrier barrier;

    public PassengerThread(int duration, CyclicBarrier barrier, String pname) {
        super(pname);
        this.duration = duration;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(duration);
            System.out.println(Thread.currentThread().getName() + " is arrived.");

            int await = barrier.await();
            if(await == 0) {
                System.out.println("Four passengers have arrived so cab is going to start.
            }

        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class ClientTest {
    public static void main(String args[]) throws InterruptedException, BrokenBarrierExcep

        System.out.println(Thread.currentThread().getName() + " has started");

        CyclicBarrier barrier = new CyclicBarrier(4);

        PassengerThread p1 = new PassengerThread(1000, barrier, "John");
        PassengerThread p2 = new PassengerThread(2000, barrier, "Martin");
        PassengerThread p3 = new PassengerThread(3000, barrier, "Joya");
        PassengerThread p4 = new PassengerThread(4000, barrier, "Sam");

        PassengerThread p5 = new PassengerThread(1000, barrier, "Pipa");
        PassengerThread p6 = new PassengerThread(2000, barrier, "Dolly");
        PassengerThread p7 = new PassengerThread(3000, barrier, "Harman");
        PassengerThread p8 = new PassengerThread(4000, barrier, "Brad");

        p1.start();
        p2.start();
```

```
31          p3.start();
32          p4.start();
33
34          p5.start();
35          p6.start();
36          p7.start();
37          p8.start();

            }  }
```

```
<terminated> ClientTest (1) [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (Feb 7, 2018, 7:04:38 AM)
main has started
main has finished
John is arrived.
Pipa is arrived.
Martin is arrived.
Dolly is arrived.
Four passengers have arrived so cab is going to start..
Joya is arrived.
Harman is arrived.
Brad is arrived.
Sam is arrived.
Four passengers have arrived so cab is going to start..
```

3. Semaphore:

Semaphore multithreading mein ek aisi utility hai jo **"Permits" (Ijazat/Tokens)** ke concept par kaam karti hai. Iska asali maqsad ye control karna hota hai ki ek waqt mein kitne threads kisi specific resource (jaise Database connection, File, ya Network socket) ko access kar sakte hain.

Ise aap ek **Parking Lot** ki tarah samajh sakte hain:

- Agar parking mein 10 jagah (permits) hain, toh 10 gaadiyan (threads) andar ja sakti hain.
- 11वीं gaadi ko tab tak bahar wait karna padega jab tak koi ek gaadi nikal na jaye.

1. Semaphore kaise kaam karta hai?

Semaphore do main methods use karta hai:

1. `acquire()` : Ye method ek permit mangta hai. Agar permit available hai, toh thread aage badh jata hai aur count 1 kam ho jata hai. Agar permit nahi hai, toh thread wahi ruk (block) jata hai.
2. `release()` : Jab thread ka kaam khatam ho jata hai, wo permit wapas kar deta hai, jisse count 1 badh jata hai aur waiting threads mein se kisi ek ko mauka milta hai.

- **Semaphore** is the new class introduced as a part of concurrency package in **JDK 1.5** .It maintains a set of permits which gets acquired by thread before using some functionality.
- It is a technique to protect your critical resource from being used by more than 'N' threads simultaneously. Semaphore maintains number of available permits. Whenever a thread wants to use some shared resource, maintained by semaphore. It asks semaphore for the permit. If permit is available thread can use the shared resource, otherwise it will wait till some other thread releases the permit or come out without using the shared resource.

Example:

```
import java.util.concurrent.Semaphore;

public class Connection {
public int number_of_connetion=0;
private static Connection INSTANCE = new Connection();
private Semaphore semaphore=new Semaphore(5, true);
public static Connection getConnection() {
    return INSTANCE;
}

public void connection(){
    try {
        semaphore.acquire();
        synchronized (this) {
            number_of_connetion++;
            System.out.println("number of conntions :"+number_of_connetion);
        }
        Thread.sleep(2000);
        synchronized (this) {
            number_of_connetion--;
            System.out.println("number of conntions :"+number_of_connetion);
        }

    } catch (Exception e) {

    }finally{
        if(semaphore!=null)
        semaphore.release();
    }
}
}
```

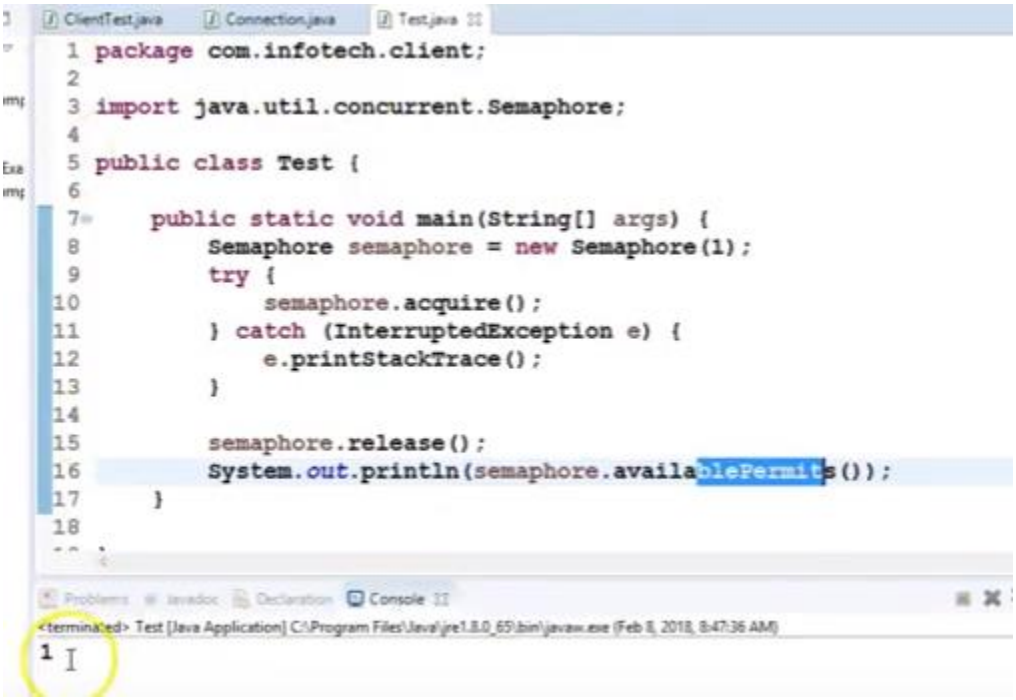
```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Semaphore {
    static ExecutorService executorService=null;

    public static void main(String[] args) {
        try {
        executorService=Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
        executorService.submit(new Runnable() {

            @Override
            public void run() {
                Connection.getConnection().connection();
            }
        });
        }
        } catch (Exception e) {
            // TODO: handle exception
        }finally{
            if(executorService!=null)
            executorService.shutdown();
        }
    }
}
```

Before Semaphore implement it will create number of connections that we are using in loop(50)

```
<terminated> Semaphore [Java Application] C:\Program Files\Java\jre7\bin\javaw.
number of conntions :1
number of conntions :2
number of conntions :3
number of conntions :4
number of conntions :5
number of conntions :6
number of conntions :7
number of conntions :8
number of conntions :9
number of conntions :10
number of conntions :9
number of conntions :8
number of conntions :7
number of conntions :6
number of conntions :5
number of conntions :4
number of conntions :3
number of conntions :2
number of conntions :1
number of conntions :0
```

After Semaphore Implement: it will create only connection pool.

```
<terminated> Semaphore [Java Application] C:\Program Files\Ja
number of conntions :1
number of conntions :2
number of conntions :3
number of conntions :4
number of conntions :5
number of conntions :4
number of conntions :3
number of conntions :4
number of conntions :3
number of conntions :4
number of conntions :3
number of conntions :2
number of conntions :3
number of conntions :4
number of conntions :5
number of conntions :4
number of conntions :3
number of conntions :2
number of conntions :1
number of conntions :0
```



Coupling and Cohesion

(A program should have – Loose coupling and High Cohesion)

Coupling

In order to create an effective application that is easy to develop, easy to maintain and easy to update, we need to know the concept of **coupling and cohesion**. **Coupling** refers to the extent to which a class knows about the other class.

There are two types of coupling -

- **Tight Coupling**(*bad programming design*)
- **Loose Coupling**(*good programming design*)

Tight Coupling

If a class A is able to access data members/instance variables of a class B **directly by dot operator**(because they were declared **public**), the two classes are said to be **tightly coupled** and it leads to *bad designing*, because all the checks made to ensure the valid access to data members of class B are bypassed by their direct access. Let's understand this by an example -

```
//Tight coupling

class Names
{
public String name;

public String getName()
{
//some code to check valid access to name
return name;
}

public void setName(String s)
{
//some code to check valid setting to name
name=s;
}
}

class ModifyData
{
public void updateName()
{
Name ob= new Name();
ob.name="Hello";   //Directly accessing name with dot operator shows tight coupling
between two classes
System.out.println(ob.name);        //Tight coupling because of bad encapsulation
}
}
```

Loose Coupling

A good application designing is creating an application with loosely coupled classes by following proper **encapsulation**, i.e. by declaring data members of a class with **private** access and forcing other classes to access them only through **public getter, setter methods**. Let's understand this by an example -

```
class Names
```

```
{
private String name;

public String getName()
{
//some code to check valid access to name
return name;
}

public void setName(String s)
{
//some code to check valid setting to name
name=s;
}
}

class ModifyData
{
public void updateName()
{
C ob= new C();
ob.setName("Howard");
ob.getName();
}
}
```

Cohesion

Cohesion refers to the extent to which a class is defined to do a **specific specialized task**. A class created with high cohesion is targeted towards a single specific purpose, rather than performing many different specific purposes.
There are two types of cohesion -

- **Low cohesion**(*bad programming design*)
- **High Cohesion**(*good programming design*)

Low Cohesion

When a class is designed to do many different tasks rather than focussing on a **single specialized task**, this class is said to be a "*low cohesive*" class. Low cohesive class are said to be *badly designed* leading to a hard time at creating, maintaining and updating them.
Let's understand this by an example -

```
class PlayerDatabase
{
public void connectDatabase();
public void printAllPlayersInfo();
public void printSinglePlayerInfo();
public void printRankings()
public void closeDatabase();
}
```

- ## High Cohesion

A good application design is creating an application with *high cohesive* classes, which are targeted towards a *specific specialized task* and such class is easy not only easy to create, but also easy to maintain and update.

```
class PlayerDatabase
{
ConnectDatabase connectD= new connectDatabase();
PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();
PrintRankings rankings = new PrintRankings();
CloseDatabase closeD= new CloseDatabase();
PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo();
}


class ConnectDatabase
{
//connecting to database.
}

class CloseDatabase
{
//closing the database connection.
}

class PrintRankings
{
//printing the players current rankings.
}

class PrintAllPlayersInfo
{
//printing all the players information.
}

class PrintSinglePlayerInfo
{
//printing a single player information.
}
```

SQL Injection in Java Code:

Preventing SQL Injection in Java is primarily about ensuring that user input is never treated as part of the SQL command itself. When you concatenate strings to build a query, the database cannot distinguish between your code and the data provided by a user.

Here are the most effective ways to secure your Java application:

1. Use PreparedStatements (The Gold Standard)

The most effective way to prevent SQL injection is using **Parameterized Queries** via the `PreparedStatement` interface. Instead of inserting values directly into the string, you use a placeholder (`?`).

The database driver treats the parameters strictly as data, not as executable code, making it impossible for a malicious string to alter the query logic.

❌ **Vulnerable Code (String Concatenation):**

Java

```
// Danger! A user could enter "' OR '1'='1" as the username
String query = "SELECT * FROM users WHERE username = '" + username + "'";
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery(query);
```

✅ **Secure Code (PreparedStatement):**

Java

```
String query = "SELECT * FROM users WHERE username = ?";
PreparedStatement pstmt = connection.prepareStatement(query);

// The driver ensures this input is escaped and treated only as a string
pstmt.setString(1, username);
ResultSet rs = pstmt.executeQuery();
```

2. Use Object-Relational Mapping (ORM)

Frameworks like **Hibernate** or **JPA (Java Persistence API)** handle SQL generation for you. If you use their standard methods, they use parameter binding under the hood.

- **JPQL Example:** Always use named parameters (starting with `:`) rather than concatenating strings inside your `createQuery` calls.

Query query = entityManager.createQuery("SELECT u FROM User u WHERE u.username = :name");

query.setParameter("name", username);

Why Char array is preferred over String for storing password?

String is immutable in java and stored in String pool. Once it's created it stays in the pool until unless garbage collected, so even though we are done with password it's available in memory for longer duration and there is no way to avoid it. It's a security risk because anyone having access to memory dump can find the password as clear text.
If we use char array to store password, we can set it to blank once we are done with it. So we can control for how long it's available in memory that avoids the security threat with String.

14) The difference between sleep and wait in Java? (answer)
Though both are used to pause currently running thread, sleep() is actually meant for short pause because it doesn't release lock, while `wait()` is meant for conditional wait and that's why it release lock which can then be acquired by another thread to change the condition on which it is waiting.

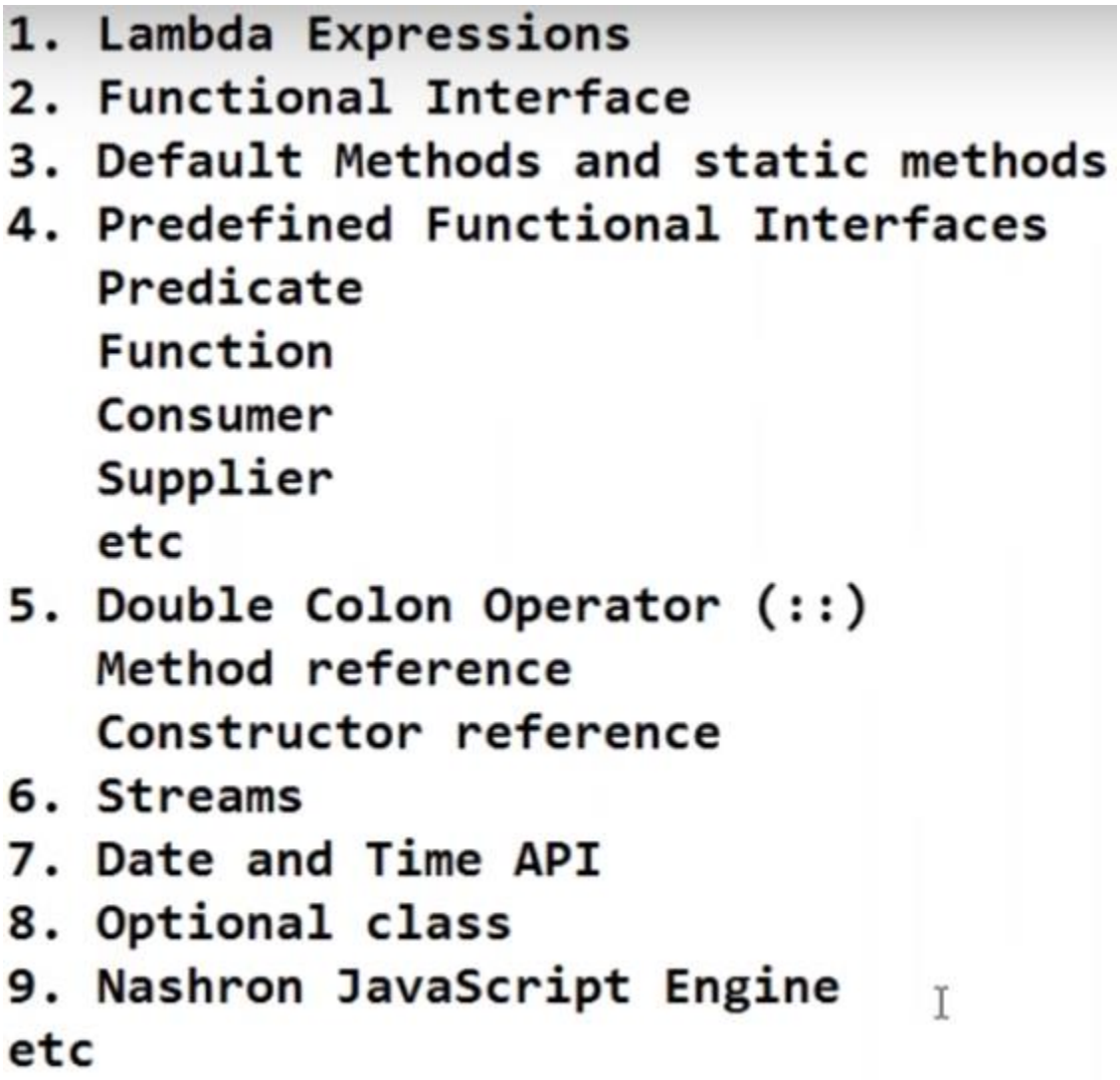
23) Is ++ operator is thread-safe in Java? (answer)
No it's not a thread safe operator because its involve multiple instructions like reading a value, incriminating it and storing it back into memory which can be overlapped between multiple threads.

61) Is it possible for two unequal objects to have the same hashcode?
Yes, two unequal objects can have same hashcode that's why collision happen in a hashmap. the equal hashcode contract only says that two equal objects must have the same hashcode it doesn't say anything about the unequal object.

Java 8 New Features:

java 8 release -18th March 2014

List of new features:



1.Lambda Expressions:

The main objective of the lambda expression is bring benefits of functional programming into Java.

Note: wherever functional interface there only we can using Lambda expression only.

i.e., If the functional interface concept is not there, then you can not using Lambda expression concept.

Lambda expression concept always associate with Functional interface only.

Lambda expression is not generalized concept, it's only specific concept which is applicable with functional interface.

What is Lambda Expression - :

a. It is an anonymous function means jiska koi naam nhhi hota, NameLess
b. Without return type – Jo kuchh bhi return nhi krta

c. Without Modifiers – Means no private,public,default,protected .

How to write Lambda Expression (With and Without Example):

Example-#1:

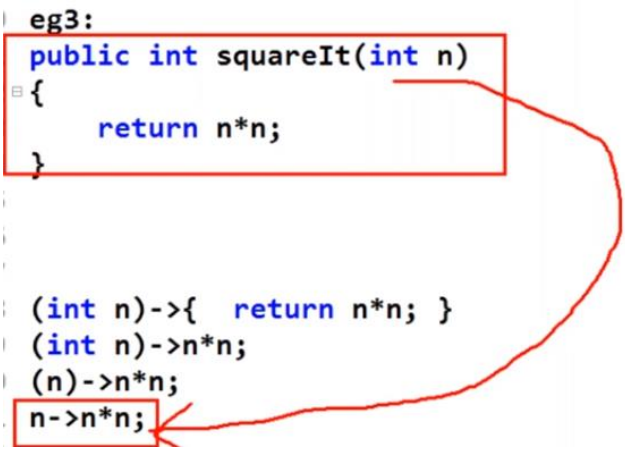
```
public void m1()
{
    System.out.println("Hello");
}


() ->{  System.out.println("Hello"); }
```

Example-#2: By passing parameters

```
eg2:
public void m1(int a,int b)
{
    System.out.println(a+b);
}


(int a,int b) ->{System.out.println(a+b);}
```

```
eg2:
public void m1(int a,int b)
{
    System.out.println(a+b);
}


(a,b) -> System.out.println(a+b);
```

Example-#3: Just add code and return:



Note: if you use return keyword you have to write inside the “ { } ”,

Note: If only one parameter then prentices “ () ” are optional.

Note:

```
n->return n*n;===>invalide
n->{return n*n;};==>valid
n->{return n*n};==>invalid
n->{n*n;};==>invalid
n->n*n;===>valid
```

```
Without curly braces we cannot use return keyword. compiler will consider returned
value automatically.
Within curly braces if we want to return some value compulsory we should use return
statement.
```

```
public void m1(String s)
{
    return s.length();
}
```

```
s -> s.length();
```

2.Functional Interface:

Functional Interface rule should contains only single abstract method.

It can contain default and static methods but should contain only one single abstract method.

Existing functional interface what we know:

```
------------------------------------------------------------

Once we write Lambda Expressions
Functional Interfaces

which contains single abstract method (SAM)

Runnable===>run()
Callable===>call()
Comparable==>compareTo()
ActionListener==>actionPerformed()


default methods & static methods
```

```
interface Interf
{
    public void m1();
    default void m2()
    {
    }
    public static void m3()
    {
    }
}
```

@FunctionalInterface Annotation:

This annotation came in java 8 to specify this interface is Functional Interface.

```
@FunctionalInterface
interface Interf
{
    public void m1();
    default void m2()
    {
    }
    public static void m3()
    {
    }
}
```

If the compile time you have specify interface as a functional interface(annotation) but is not follow the rule then it will give you error at compile time-

```
@FunctionalInterface
interface Interf
{
    default void m2()
    {
    }
    public static void m3()
    {
    }
}
```

Compile Test.java-

```
D:\durgaclasses>javac Test.java
Test.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Interf is not a functional interface
    no abstract method found in interface Interf
1 error

D:\durgaclasses>
```

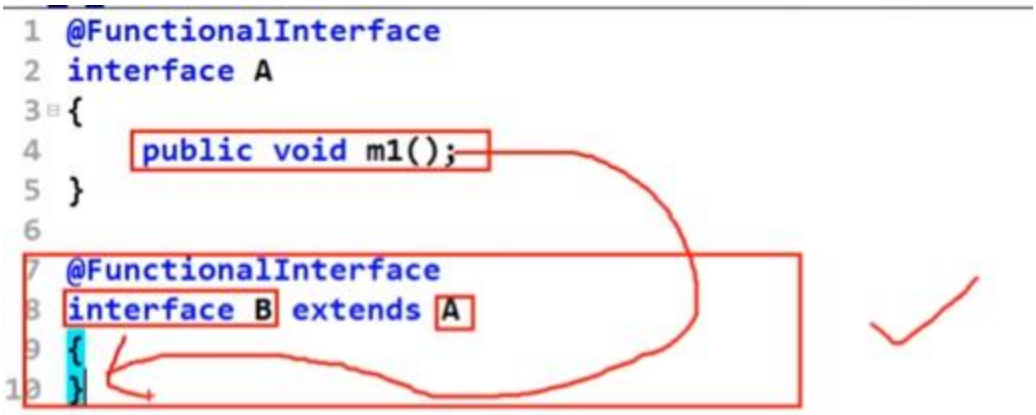
Example 2-

Compile Test.java-

```
@FunctionalInterface
interface Interf
{
    public void m1();
    public void m11();
    default void m2()
    {
    }
    public static void m3()
    {
    }
}
```

```
D:\durgaclasses>javac Test.java
Test.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Interf is not a functional interface
    multiple non-overriding abstract methods found in interface Interf
1 error

D:\durgaclasses>
```

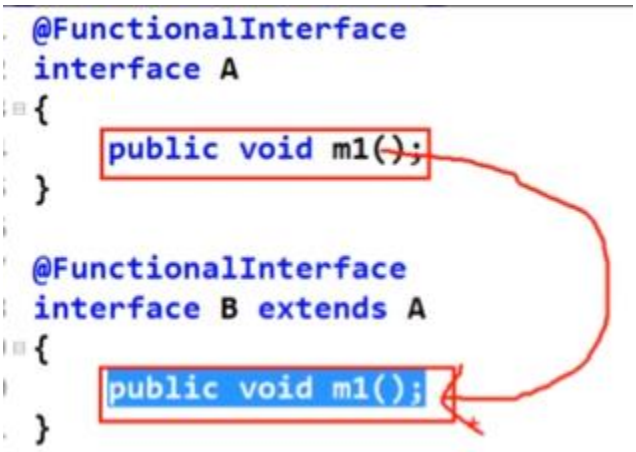
Functional Interface with respect to inheritance-:



Above example is valid because m1() by default to available for B interface.

Compile Test.java-

Example 2- also Valid



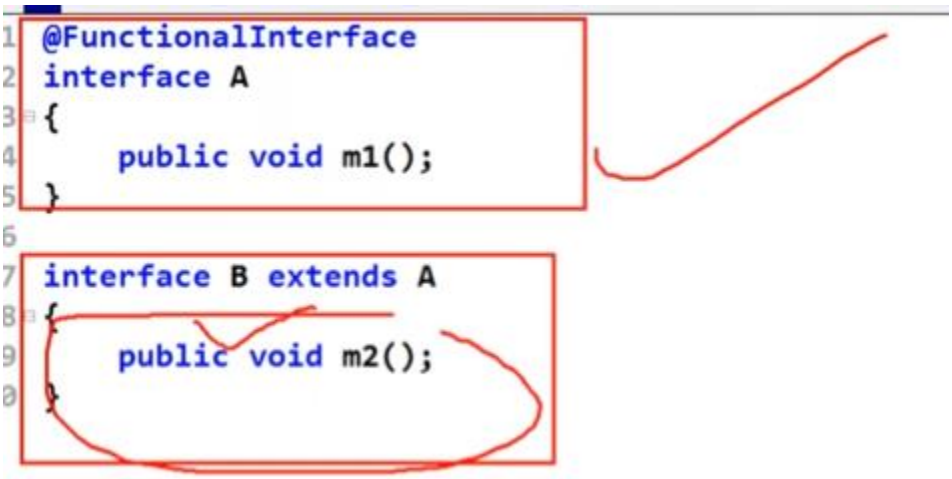
Example 3- Invalid

```java
@FunctionalInterface
interface A
{
    public void m1();
}

@FunctionalInterface
interface B extends A
{
    public void m2();
}
```

Compile Test.java (because Functional interface contains only simple abstract method – one is coming from parent interface (m1) and second you have specify in B (m2)).

```
D:\durgaclasses>javac Test.java
Test.java:7: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  B is not a functional interface
    multiple non-overriding abstract methods found in interface B
1 error

D:\durgaclasses>
```

Example 4- Valid



Note – above case is valid because here we are not using annotation e.i, B interface is a normal interface not the functionalInterface (missing annotation).

Compile Test.java

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>
```

Lambda Expression with Functional Interface-:

without Lambda Expression:

Specification for above - Demo.java class is implementations for parent interface(InterF)

```
interface Interf
{
    public void m1();
}
class Demo implements Interf
{
    public void m1()
    {
        System.out.println("Hello...");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Demo d= new Demo();
        d.m1();
    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Hello...

D:\durgaclasses>
```

With Lambda Expression- you can implement m1() methods of InterF interace using Lambda Expression, we do not need to implement top level separate class(Demo.java)-

```
interface Interf
{
    public void m1();
}
class Test
{
    public static void main(String[] args)
    {
        Interf i= ()->System.out.println("Hello...By Lambda Expression");
        i.m1();
    }
}
```

Output-

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Hello...By Lambda Expression

D:\durgaclasses>
```

Note: lines of the code reduced

Example-#2 – calling m1() multiple times

```
public static void main(String[] args)
{
    Interf i= ()->System.out.println("Hello...By Lambda Expression");
    i.m1();
    i.m1();
    i.m1();
    i.m1();
    i.m1();
}
```

Output-

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Hello...By Lambda Expression
Hello...By Lambda Expression
Hello...By Lambda Expression
Hello...By Lambda Expression
Hello...By Lambda Expression

D:\durgaclasses>
```

Example-#3: just returning then

```
interface Interf
{
    public int squareIt(int n);
}
class Test
{
    public static void main(String[] args)
    {
        Interf i = n->n*n;
        System.out.println(i.squareIt(4));
        System.out.println(i.squareIt(5));
    }
}
```

Lambda Expression in Collection Concept:

Example-#1:

Without lambda expression:

```java
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Integer> l= new ArrayList<Integer>();
        l.add(20);
        l.add(10);
        l.add(25);
        l.add(5);
        l.add(30);
        l.add(0);
        l.add(15);
        System.out.println(l);
        Collections.sort(l,new MyComparator());
        System.out.println(l);
    }
}
```

```java
import java.util.*;
class MyComparator implements Comparator<Integer>
{
    public int compare(Integer I1,Integer I2)
    {
        return (I1<I2)?-1:(I1>I2)?1:0;
    }
}
```

With Lambda Expression:

Comparator(functional interface) contain only single abstract method(compare()).

```
Comparator
int compare(Object obj1,Object obj2)
  returns -ve iff obj1 has to come before obj2
  returns +ve iff obj1 has to come after obj2
  returns 0 iff obj1 and obj2 are equal
```

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Integer> l= new ArrayList<Integer>();
        l.add(20);
        l.add(10);
        l.add(25);
        l.add(5);
        System.out.println(l);
        Comparator<Integer> c=(I1,I2)-> (I1<I2)?-1:(I1>I2)?1:0;
        Collections.sort(l,c);
        System.out.println(l);
    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
[20, 10, 25, 5]
[5, 10, 20, 25]

D:\durgaclasses>
```

Anonymous inner Class vs Lambda expression:

Note: lambda expression is not the replacement of anonymous inner class.

Anonymous inner class- the class without having name such type of inner class are called anonymous inner class.

```
Thread t = new Thread()
{
    ...
};
```

We are writing a class that extends Thread class

Similarly:

```
Runnable r= new Runnable()
{
   ...
};
```

Anonymous inner class example:

```
class Test
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            public void run()
            {
                for(int i =0; i<10;i++)
                {
                    System.out.println("Child Thread");
                }
            }
        };
        Thread t = new Thread(r);
        t.start();
        for(int i =0; i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
```

Anonymous inner class with Lambda Expression:

```
class Test
{
    public static void main(String[] args)
    {
        Runnable r=()->{
            for(int i =0; i<10; i++)
            {
                System.out.println("Child Thread");
            }
        };
        Thread t = new Thread(r);
        t.start();
        for(int i =0; i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
}
```

Note: Lambda expression we can not use for every anonymous inner class:

Like – normal anonymous inner class – below case possible for this but not possible with lambda expression , because lambda expression only applicable for single abstract class.

```
interface A
{
   m1();
   m2();
}
```

```
A a = new A()
{
    public void m1()
    {
    }
    public void m2()
    {
    }
};
```

```
Anonymous Inner class != Lambda Expressions

If anonymous Inner class implements an interface that contains single abstract method
then only we can replace that anonymous inner class with lambda expressions
```

```
Anonymous Inner class can extend a normal class
Anonymous Inner class can extend an abstract class
Anonymous Inner class can implement an interface which contains any number of
abstract methods

Lambda Expression can implement an interface which contains a single abstract
method(FI)

Anonymous Inner class != Lambda Expression
Anonymous Inner class > Lambda Expression
-------------------------------------------------------
```

3.Default method and Static Method:

Default method (virtual extension method or defender method)and Static Method:

```
Default Methods:
----------------
until 1.7V:
----------
Every method present inside interface is always: public and abstract

    void m1();
    public void m1();
    abstract void m1();
    public abstract void m1();

Related Methods:
   1.8 V: default methods+static methods
   1.9 V: private methods
```

Note: 1.8 v we can declare default implementation of a method.

```
Without effecting implementation classes if we want to add new method to the
interface===>Default Methods
```

Example-1:

```java
interface Interf
{
    default void m1()
    {
        System.out.println("Default Method");
    }
}
class Test implements Interf
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
```

Output:

Static Methods inside interface:

From java 1.8 onward you can declare static methods inside interface.

Note: interface static methods only call using interface name only.

Example 1 :

```java
interface Interf
{
    public static void m1()
    {
        System.out.println("Interface Static Method");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Interf.m1();//1
        //m1();//2
        //Test.m1();//3
        //Test t = new Test();
        //t.m1();//4

    }

}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Interface Static Method
```

4. Predefined functional interface:

Introduced in java 1.8 to make the lambda expression as the common coding activity.

```
Predefined Functional Interfaces:
---------------------------------
Predicate
Function
Consumer
Supplier
--------------
Two argument Predefined functional interfaces:

BiPredicate
BiFunction
BiConsumer
-----------------------
primitive Functional interfaces
---------------------------
IntPredicate
IntFunction
IntConsumer
....
```

1. Predicate Interface:

```
Predicate(I):
------------
public abstract  boolean  test(T t)
```

Note: predicate function is use for conditional checking. Predicate function return type always be Boolean-

```
Predicate<T>     --->boolean
```

Note: Predicate always take input type and return Boolean result.

Example with and without lambda expression.

Example-#1 – Integer

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Predicate<Integer> p1= i->i%2==0;
        System.out.println(p1.test(10));
        System.out.println(p1.test(15));

    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
true
false

D:\durgaclasses>
```

Example-#2 – String

Write a Predicate to check whether length of String is > 5 or not.

```
Predicate<String> p=s->s.length()>5;
```

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        String[] s ={"Nag","Chiranjeevi","Venkatesth","Balaiah","Sunny","Katrina"};
        Predicate<String> p=s1->s1.length()>5;
        for(String s1 : s)
        {
            if(p.test(s1))
            {
                System.out.println(s1);
            }
        }

    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Chiranjeevi
Venkatesth
Balaiah
Katrina

D:\durgaclasses>
```

Example-#3 – Custom Class (Employee)

```java
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Employee> l= new ArrayList<Employee>();
        l.add(new Employee("Durga",1000));
        l.add(new Employee("Ravi",2000));
        l.add(new Employee("Shiva",3000));
        l.add(new Employee("Mahesh",4000));
        l.add(new Employee("Adarsh",5000));
        l.add(new Employee("Sagar",6000));

        Predicate<Employee> p = e -> e.salary>3000;

    }
}
```

```java
Predicate<Employee> p = e -> e.salary>3000;
for(Employee e1 : l)
{
    if(p.test(e1))
    {
        System.out.println(e1.name+":"+e1.salary);
    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Mahesh:4000.0
Adarsh:5000.0
Sagar:6000.0

D:\durgaclasses>
```

Predicate Join:

```
p1--->to check whether the number is even or not
p2--->to check whether the number >10 or not

p1.and(p2).test(34)


Predicate Joining:
-----------------
p1.or(p2)

p1.negate()
```

Example 1:

```
public static void main(String[] args)
{
    int[] x ={0,5,10,15,20,25,30,35};
    Predicate<Integer> p1 = i->i%2==0;
    Predicate<Integer> p2 = i-> i>10;
    // and(),or(),negate()
    System.out.println("The numbers which are even and > 10 are:");
    for(int x1 : x)
    {
        if(p1.and(p2).test(x1))
        {
            System.out.println(x1);
        }
    }
}
```

Output:

```
D:\durgaclasses>java Test
The numbers which are even and > 10 are:
20
30

D:\durgaclasses>_
```

Example -2 : to check or condition.

```
public static void main(String[] args)
{
    int[] x ={0,5,10,15,20,25,30,35};
    Predicate<Integer> p1 = i->i%2==0;
    Predicate<Integer> p2 = i-> i>10;
    // and(),or(),negate()
    System.out.println("The numbers which are even or > 10 are:");
    for(int x1 : x)
    {
        if(p1.or(p2).test(x1))
        {
            System.out.println(x1);
        }
    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
The numbers which are even or > 10 are:
0
10
15
20
25
30
35

D:\durgaclasses>
```

Example 4: to check negative

```
public static void main(String[] args)
{
    int[] x ={0,5,10,15,20,25,30,35};
    Predicate<Integer> p1 = i->i%2==0;
    Predicate<Integer> p2 = i-> i>10;
    // and(),or(),negate()
    System.out.println("The numbers which are not even :");
    for(int x1 : x)
    {
        if(p1.negate().test(x1))
        {
            System.out.println(x1);
        }
    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
The numbers which are not even :
5
15
25
35

D:\durgaclasses>
```

2. Function Functional Interface:

Input (T) and return (R) with any type.

```
Function   functional interface
interface Function<T,R>
{
   public R  apply(T t);
}
```

Example-#1 – Take input(Integer) and return integer:

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Function<Integer,Integer> f=i->i*i;
        System.out.println(f.apply(4));


    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
16

D:\durgaclasses>
```

Example-#2 – Input String and return Integer:

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Function<String,Integer> f=s->s.length();
        System.out.println(f.apply("durga"));
    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
5

D:\durgaclasses>
```

Example-#3 – Input String and return String:

```java
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Function<String,String> f=s->s.toUpperCase();
        System.out.println(f.apply("durgasoftwaresolutions"));
    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
DURGASOFTWARESOLUTIONS

D:\durgaclasses>
```

Function Chaining:

Two function can be combine to form more complex functions.

```
Function chaining:
------------------
f1.andThen(f2).apply(i)

first f1 followed by f2

f1.compose(f2).apply(i)
first f2 and then f1
```

Example 1: first f1 and then f2:

```java
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Function<Integer,Integer> f1=i->2*i;
        Function<Integer,Integer> f2=i->i*i*i;
        System.out.println(f1.andThen(f2).apply(2));
    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
64

D:\durgaclasses>
```

Example 2: first calculate f2 and then compose result with f1:

```
import java.util.function.*;
class Test
{
    public static void main(String[] args)
    {
        Function<Integer,Integer> f1=i->2*i;
        Function<Integer,Integer> f2=i->i*i*i;
        System.out.println(f1.andThen(f2).apply(2));
        System.out.println(f1.compose(f2).apply(2));
    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
64
16

D:\durgaclasses>
```

3. Consumer Functional Interface:

Consumer always take some input type but do not return any values means it's void type.

```
Consumer<T>    --->void
```

That is :

```
Predicate<T>     --->boolean
Function<T,R>    --->R type
Consumer<T>    --->void
```

```
interface Consumer<T>
{
    public void accept(T t);
}
```

Example 1-

```
import java.util.function.*;
class Movie
{
     String name;
     Movie(String name)
     {
          this.name=name;
     }
}

class Test
{
    public static void main(String[] args)
    {
        Consumer<Movie> c1= m->System.out.println(m.name+" ready to release");
        Consumer<Movie> c2= m->System.out.println(m.name+" released but it is bigger f
        Consumer<Movie> c3= m->System.out.println(m.name+" storing information in data

        Movie m= new Movie("Spider");
        c1.accept(m);


    }
}
```

Output:

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Spider ready to release

D:\durgaclasses>
```

4. Supplier Functional Interface:

Supplier will not take any input.

Just supply my required objects and it won't take any input-->Supplier

```
interface Supplier<R>
{
    public R get();
}
```

```
import java.util.function.*;
import java.util.Date;
class Test
{
    public static void main(String[] args)
    {
        Supplier<Date> s=()->new Date();
        System.out.println(s.get());
        System.out.println(s.get());
        System.out.println(s.get());
        System.out.println(s.get());

    }
}
```

```
D:\durgaclasses>javac Test.java

D:\durgaclasses>java Test
Tue Aug 14 10:58:15 IST 2018
Tue Aug 14 10:58:15 IST 2018
Tue Aug 14 10:58:15 IST 2018
Tue Aug 14 10:58:15 IST 2018

D:\durgaclasses>
```

5.Double Colon Operator(::) :

Method and Constructor Reference(::):

```
classname::method name
object reference::method name
```

Java 8 mein **Method Reference** (jise hum **Double Colon** `::` **operator** kehte hain) ek aisa feature hai jo Lambda Expression ko aur bhi chota aur readable bana deta hai.

Asal mein, Method Reference ek **shorthand** (chota raasta) hai kisi existing method ko bulaane ka.

Method Reference Kaun Si Problem Solve Karta Hai?

Maan lijiye aap ek Lambda Expression likh rahe hain jo sirf ek existing method ko call kar raha hai:

- **Lambda:** `str -> System.out.println(str)`
- **Method Reference:** `System.out::println`

Dono ka kaam bilkul same hai, lekin `::` wala tarika zyada saaf dikhta hai.

```
class Test
{
    public static void m1()
    {
        for(int i =0; i<10;i++)
        {
            System.out.println("Child Thread-1");
        }
    }

    public static void main(String[] args)
    {
        Runnable r = Test::m1;
        Thread t = new Thread(r);
        t.start();
        for(int i =0; i<10;i++)
        {
            System.out.println("Main Thread-1");
        }
    }
}
```

Notes: above example is for static method.

Example 2: for non-static method-

```java
class Test
{
    public void m1()
    {
        for(int i =0; i<10;i++)
        {
            System.out.println("Child Thread-1");
        }
    }

    public static void main(String[] args)
    {
        Test t1= new Test();
        Runnable r = t1::m1;
        Thread t = new Thread(r);
        t.start();
        for(int i =0; i<10;i++)
        {
            System.out.println("Main Thread-1");
        }
    }
```

1. Static Method Reference

Agar aap kisi class ke static method ko call karna chahte hain.

- **Syntax:** `ClassName::methodName`
- **Example:**

```
Java
```

```java
// Lambda
(a, b) -> Math.max(a, b);
// Method Reference
Math::max;
```

2. Instance Method Reference (Specific Object)

Jab aapke paas pehle se ek object ho aur aap uska method call karna chahein.

- **Syntax:** `objectReference::methodName`
- **Example:**

```
Java
```

```java
String s = "hello";
// Lambda
() -> s.length();
// Method Reference
s::length;
```

3. Instance Method Reference (Arbitrary Object of a Type)

Ye tab use hota hai jab aap kisi class ke saare objects par wo method chalana chahte hain.

- **Syntax:** `ClassName::methodName`
- **Example:**

Java

```
List<String> list = Arrays.asList("a", "b", "c");
// Lambda
list.forEach(s -> s.toUpperCase());
// Method Reference
list.forEach(String::toUpperCase);
```

4. Constructor Reference

Naya object banane ke liye.

- **Syntax:** `ClassName::new`
- **Example:**

Java

```
// Lambda
() -> new ArrayList<>();
// Method Reference
ArrayList::new;
```

Comparison Table: Lambda vs Method Reference

Scenario	Lambda Expression	Method Reference
Static Method	`(n) -> Math.abs(n)`	`Math::abs`
Instance Method	`(s) -> s.toLowerCase()`	`String::toLowerCase`
Constructor	`() -> new Employee()`	`Employee::new`
Printing	`(x) -> System.out.println(x)`	`System.out::println`

6. Stream API:

If you want process of the object from collection then you can use Steams concept.

It’s an Interface present in java.util.steam package.

Below method we will discuss:

1. ***Steam()***
2. ***Filter()***
3. ***Map()***
4. ***Collect()***
5. ***Count()***
6. ***Sorted()***
7. ***Min()***
8. ***Max()***
9. ***ForEach()***
10. ***toArray()***

```java
import java.util.stream.Stream;


public class ForEach_Main {

    public static void main(String[] args) {

        Stream<Integer> stream=Stream.of(1,2,3,4);
        System.out.println("***** Normal Stream Object *****");
        stream.forEach(p->System.out.println(p));

        List<Integer> list=new ArrayList<>(Arrays.asList(1,2,3,4));

        stream=list.stream();
        System.out.println("***** List Stream Object *****");
        stream.forEach(p->System.out.println(p));

    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe
***** Normal Stream Object *****
1
2
3
4
***** List Stream Object *****
1
2
3
4
```

Stream Intermediate Operator:

Intermediate operations return the stream itself so you can chain multiple method calls in a row. Let's learn important ones.

Java **Stream API** mein **Intermediate Operations** wo operations hote hain jo ek Stream ko process karke doosri Stream return karte hain. Inki sabse badi khasiyat ye hai ki ye **Lazy** hote hain—yani jab tak aap koi *Terminal Operation* (jaise `collect()`, `forEach()`) nahi chalate, tab tak ye execute nahi hote.

Inhe aap ek "Pipe" ki tarah samajh sakte hain jisme se data guzarta hai aur filter ya transform hota hai.

filter(Predicate):

Filter accepts a predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. forEach) on the result.

Ye elements ko kisi condition ke base par filter karta hai.

- **Example:** Sirf wo naam nikalo jo "A" se shuru hote hain.
- `list.stream().filter(s -> s.startsWith("A"))`

```java
public class ForEach_Main {

    public static void main(String[] args) {

        List<String> memberNames = new ArrayList<>();
        memberNames.add("Amitabh");
        memberNames.add("Rahul");
        memberNames.add("Aman");
        memberNames.add("Salman");
        memberNames.add("Lokesh");

        System.out.println("***** First Way *****");
        Stream<String> stream=memberNames.stream();
        Predicate<String> predicate=s->s.startsWith("A");
        stream=stream.filter(predicate);
        stream.forEach(System.out::println);


        System.out.println("***** Second Way *****");
        memberNames.stream().filter((s)->s.startsWith("A")).forEach(System.out::println);

    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (Feb 19, 2019, 12:52:57 PM)
***** First Way *****
Amitabh
Aman
***** Second Way *****
Amitabh
Aman
```

With Stream Concept: ***filter() method of streams. -> filter(predicate) means getting input and returning Boolean value.***

```java
package com.java8.examples;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<Integer> list=new ArrayList<Integer>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(25);
        System.out.println("Number List :"+list);
        List<Integer> new_list=new ArrayList<Integer>();

        new_list=list.stream().filter(i->i%2==0).collect(Collectors.toList());


        System.out.println("Even Number New List :"+new_list);
    }

}
```

```
<terminated> Main [Java Application] C:\Program
Number List :[0, 5, 10, 15, 20, 25]
Even Number New List :[0, 10, 20]
```

Example-2*: write a program to make double of all elements in a list and insert into another New list.*

Without Streams Concept:

```java
package com.java8.examples;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<Integer> list=new ArrayList<Integer>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(25);
        System.out.println("Number List :"+list);
        List<Integer> new_list=new ArrayList<Integer>();

        for(Integer i: list) {

            new_list.add(i*2);
        }

        System.out.println("Even Number New List :"+new_list);
    }

}
```

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1
Number List :[0, 5, 10, 15, 20, 25]
Even Number New List :[0, 10, 20, 30, 40, 50]
```

<u>With Steams Concept: map() method</u> -> map(function) means getting input and retuning output

```java
    public void streamMap() {
        java.util.List<Integer> list = new ArrayList<>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(25);
        java.util.List<Integer> newList = new ArrayList<>();
        newList = list.stream().map(i->i+5).collect(Collectors.toList());
        System.out.println(newList);

    }
```

```
<terminated> Main [Java Applica
[5, 10, 15, 20, 25, 30]
```

```
Collection and Stream:
----------------------
If we want to represent a ground of objects as a single entity then we should go for
collection.

If we want to process objects from the collection-Stream

Stream s = c.stream();
java.util.stream pkg
```

```
filter(Predicate)
map(Function)


list of marks
filter based on marks: select only passed students
marks.stream().filter(m->m>=35)

5 grace marks


failedstudents.stream().map(i->i+5)
```

```
filter:
-------
input-->10 elements
output-->0 to 10   or  <=10

map:
---
input:10 elements
output: 10
```

In above screenshot if filter have 10 input elements then output will be 0 to 10 or equal to 10.

But in map() if input 10 element then output compulsory will be 10,because for every element will

map(Function):

Java Stream API mein map() ek bahut hi powerful intermediate operation hai. Iska kaam hota hai Transformation.

Ye har element ko kisi doosre form mein convert (transform) kar deta hai.

- **Example:** Saare strings ko UpperCase mein badalna.
- `list.stream().map(String::toUpperCase)`

Basic Example: String to Length (Integer)

Maan lijiye aapke paas names ki list hai aur aapko unki lengths chahiye.

Java

```
ort java.util.*;
ort java.util.stream.*;

lic class MapExample {
  public static void main(String[] args) {
      List<String> names = Arrays.asList("Java", "Spring", "Hibernate");

      // map() ka use karke String ko Integer (length) mein convert kar rahe hain
      List<Integer> nameLengths = names.stream()
                                       .map(String::length) // Method Reference
                                       .collect(Collectors.toList());

      System.out.println(nameLengths); // Output: [4, 6, 9]
  }
```

Real-World Example: Object Transformation

Aksar humare paas Database se `Entity` class aati hai, lekin hum use `DTO` (Data Transfer Object) mein convert karke UI par bhejna chahte hain.

Java

```java
class User {
    String name;
    String email;
    String password; // Hum password nahi dikhana chahte
    // Constructor and getters...
}

class UserDTO {
    String name;
    String email;
    public UserDTO(String name, String email) { this.name = name; this.email = em
    @Override public String toString() { return name + " (" + email + ")"; }
}

// Transformation logic
List<UserDTO> dtoList = users.stream()
    .map(user -> new UserDTO(user.getName(), user.getEmail())) // User object ->
    .collect(Collectors.toList());
```

Key Point to Remember

`map()` **vs** `flatMap()` **:**

- `map()` : Ek element ke badle **ek** hi transformed element deta hai. (1-to-1 mapping)
- `flatMap()` : Ek element ke badle **multiple** elements (stream) de sakta hai. (1-to-many mapping)

sorted():

```
public class ForEach_Main {

    public static void main(String[] args) {

        List<String> memberNames = new ArrayList<>();
        memberNames.add("Amitabh");
        memberNames.add("Rahul");
        memberNames.add("Aman");
        memberNames.add("Salman");
        memberNames.add("Lokesh");

       memberNames.stream().sorted().map(String::toUpperCase).forEach(System.out::println);

    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (Feb 19, 2019, 12:57:42 PM)
AMAN
AMITABH
LOKESH
RAHUL
SALMAN
```

Example-4: *sorted() method*

```
1  package com.java8.examples;
2
3  import java.util.ArrayList;
6
7  public class Main {
8
9      public static void main(String[] args) {
10
11         List<Integer> list=new ArrayList<Integer>();
12         list.add(0);
13         list.add(5);
14         list.add(10);
15         list.add(15);
16         list.add(20);
17         list.add(25);
18         System.out.println("Number List :"+list);
19         List<Integer> new_list=new ArrayList<Integer>();
20
21         new_list=list.stream().sorted().collect(Collectors.toList());
22
23         System.out.println("Even Number New List :"+new_list);
24
25     }
26
27 }
```

```
<terminated> Main [Java Application] C:\Program Files\Java\jre
Number List :[0, 5, 10, 15, 20, 25]
Even Number New List :[0, 5, 10, 15, 20, 25]
```

Example-5: *manually sorting: using comparator:*

```
Comparator-->compare(obj1,obj2)
    returns -ve iff obj1 has to come before obj2
    returns +ve iff obj1 has to come after obj2
    returns 0 iff obj1 & obj2 are equal
```

```
(i1,i2)->(i1<i2)?1:(i1>i2):-1:0
```

sorted()==>According to default natural sorting order
sorted(Comparator)==>Customized sorting

```
1  package com.java8.examples;
2
3  import java.util.ArrayList;
6
7  public class Main {
8
9      public static void main(String[] args) {
10
11         List<Integer> list=new ArrayList<Integer>();
12         list.add(0);
13         list.add(5);
14         list.add(10);
15         list.add(15);
16         list.add(20);
17         list.add(2);
18         System.out.println("Number List :"+list);
19         List<Integer> new_list=new ArrayList<Integer>();
20
21         new_list=list.stream().sorted((i,i2)->(i>i2)?1:(i<i2)?-1:0).collect(Collectors.toL
22
23         System.out.println("Even Number New List :"+new_list);
24
25     }
26
27 }
```

```
<terminated> Main [Java Application] C:\Program Files\Java\
Number List :[0, 5, 10, 15, 20, 2]
Even Number New List :[0, 2, 5, 10, 15, 20]
```

***Example-6:* For string compare:**

(s1,s2)->s1.compareTo(s2) ===>Natural Sorting Order
(s1,s2)->s2.compareTo(s1) ==>reverse of natural sorting order
(s1,s2)->-s1.compareTo(s2) ==>reverse of natural sorting order

```
7  public class Main {
8
9      public static void main(String[] args) {
10
11         List<Integer> list=new ArrayList<Integer>();
12         list.add(0);
13         list.add(5);
14         list.add(10);
15         list.add(15);
16         list.add(20);
17         list.add(2);
18
19         List<String> slist=new ArrayList<String>();
20         slist.add("Sunny");
21         slist.add("Kajal");
22         slist.add("Prabhas");
23         slist.add("Anushka");
24         slist.add("Mallika");
25
26         System.out.println("String List :"+slist);
27         List<String> new_list=new ArrayList<String>();
28
29         new_list=slist.stream().sorted((s1,s2)->s1.compareTo(s2)).collect(Collectors.toList());
30
31         System.out.println("Even Number New List :"+new_list);
32
33     }
34
35 }
```

Console

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (21-Apr-2020, 3:45:00 PM)
String List :[Sunny, Kajal, Prabhas, Anushka, Mallika]
Even Number New List :[Anushka, Kajal, Mallika, Prabhas, Sunny]
```

```
    l.add("AAA");
    System.out.println(l);
    Comparator<String> c= (s1,s2)->{
            int l1=s1.length();
            int l2=s2.length();
            if(l1<l2) return -1;
            else if(l1>l2) return +1;
            else return s1.compareTo(s2);
        };
    List<String> sortedList=l.stream().sorted(c).collect(Collectors.toList());
    System.out.println(sortedList);
```

Min() & max():

Exampl-7:min() and max() :

```
min(Comparator)
max(Comparator)

[5,3,10,15,4]
[3,4,5,10,15]==>ASceding Order

[15,10,5,4,3]==>Descending order
```

Min() example:

```java
package com.java8.examples;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<Integer> list=new ArrayList<Integer>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(2);

        List<String> new_list=new ArrayList<String>();
        List<Integer> new_list1=new ArrayList<Integer>();

        Integer a=list.stream().min((i1,i2)->i1.compareTo(i2)).get();


        System.out.println("Even Number New List :"+a );

    }

}
```

Console

<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (21-Apr-2020, 3:

```
Even Number New List :0
```

Max() example:

```java
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<Integer> list=new ArrayList<Integer>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(2);

        List<String> new_list=new ArrayList<String>();
        List<Integer> new_list1=new ArrayList<Integer>();

        Integer a=list.stream().max((i1,i2)->i1.compareTo(i2)).get();


        System.out.println("Even Number New List :"+a );

    }

}
```

Console

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (21-Apr-2020, 3
Even Number New List :20
```

Example-8: *Iterate collection elements using Streams concept:*

```java
for(Integer i1: l)
{
   System.out.println(i1);
}

l.stream().forEach(Function)

l.stream().forEach(System.out::println)
```

```java
    list.stream().forEach(System.out::println);
}
```

void java.util.stream.Stream.forEach(Consumer<? super Integer> action)

```java
package com.java8.examples;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<Integer>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(2);

        list.stream().forEach(System.out::println);

    }
}
```

```
<termin
0
5
10
15
20
2
```

```java
package com.java8.examples;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<Integer>();
        list.add(0);
        list.add(5);
        list.add(10);
        list.add(15);
        list.add(20);
        list.add(2);

        Consumer<Integer> fun=i->{
            System.out.println("squer of :"+i+" is :"+(i*i));
        };
        list.stream().forEach(fun);

    }

}
```

```
<terminated> Main [Java App
squer of :0 is :0
squer of :5 is :25
squer of :10 is :100
squer of :15 is :225
squer of :20 is :400
squer of :2 is :4
```

toArray():

***Example-9:* toArray():**

toArray()
To convert stream of objects into array

Method and constructor reference:

```
Integer[] i=l.stream().toArray(Integer[]::new);
```

```java
public static void main(String[] args)
{
    ArrayList<Integer> l = new ArrayList<Integer>();
    l.add(10);
    l.add(0);
    l.add(15);
    l.add(5);
    l.add(20);
    Integer[] i=l.stream().toArray(Integer[]::new);
    for(Integer i1 : i)
    {
        System.out.println(i1);
    }
}
```

```
D:\durgaclasses>java Test
10
0
15
5
20

D:\durgaclasses>
```

```java
package com.java8.examples;

import java.util.stream.Stream;

public class Main {

    public static void main(String[] args) {

        Stream<Integer> s=Stream.of(9,99,20,349);
        s.forEach(System.out::println);

    }

}
```

```
<termir
9
99
20
349
```

```java
public static void main(String[] args)
{
    Stream<Integer> s= Stream.of(9,99,999,9999,99999);
    s.forEach(System.out::println);

    Integer[] i={10,20,30,40,50};
    Stream.of(i).forEach(System.out::println);

}
```

7.Date and Time API:

it is also called Joda Time API because it's developed by Joda.org.

With respect to conventions and performance wise it's good.

Package name for this API is – java.time*

Example-1:

```java
package com.java8.examples;

import java.time.LocalDate;
import java.time.LocalTime;

public class Main {

    public static void main(String[] args) {

        LocalDate ld=LocalDate.now();
        LocalTime lt=LocalTime.now();

        System.out.println("Current Date :"+ld);
        System.out.println("Current Time :"+lt);

    }

}
```

```
<terminated> Main [Java Application]
Current Date :2020-04-21
Current Time :22:18:37.933
```

Example-2: print date according our requirement.

```java
package com.java8.examples;

import java.time.LocalDate;

public class Main {

    public static void main(String[] args) {

        LocalDate ld = LocalDate.now();

        int dd = ld.getDayOfMonth();
        int mm = ld.getMonthValue();
        int yyyy = ld.getYear();

        System.out.printf("%d-%d-%d", dd, mm, yyyy);

    }

}
```

```
<terminated> Mai
21-4-2020
```

Example-3: print time according our requirement:

```java
package com.java8.examples;

import java.time.LocalTime;

public class Main {

    public static void main(String[] args) {

        LocalTime ld = LocalTime.now();

        int hh = ld.getHour();
        int mm = ld.getMinute();
        int ss = ld.getSecond();

        System.out.printf("%d:%d:%d", hh, mm, ss);

    }

}
```

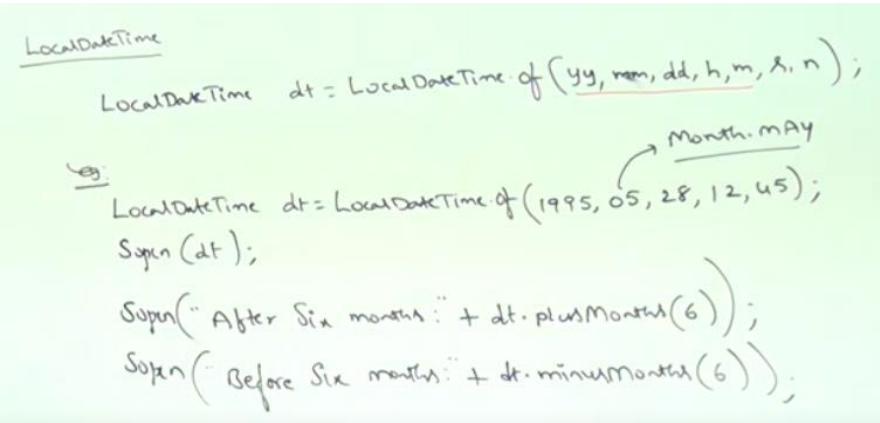
```
<terminated
22:43:48
```

```java
package com.java8.examples;

import java.time.LocalDateTime;

public class Main {

    public static void main(String[] args) {

        LocalDateTime ld = LocalDateTime.now();

        int dd = ld.getHour();
        int mm = ld.getMonthValue();
        int yyyy = ld.getYear();

        int hh = ld.getHour();
        int m = ld.getMinute();
        int ss = ld.getSecond();
        System.out.println("Date : ========");
        System.out.printf("%d-%d-%d", dd, mm, yyyy);
        System.out.println();
        System.out.println("Time : ========");
        System.out.printf("%d-%d-%d", hh, m, ss);

    }

}
```

```
<terminated> Main [Java
Date : ========
22-4-2020
Time : ========
22-47-55
```

<u>Example-4:</u> How to represent both date and time by providing date:

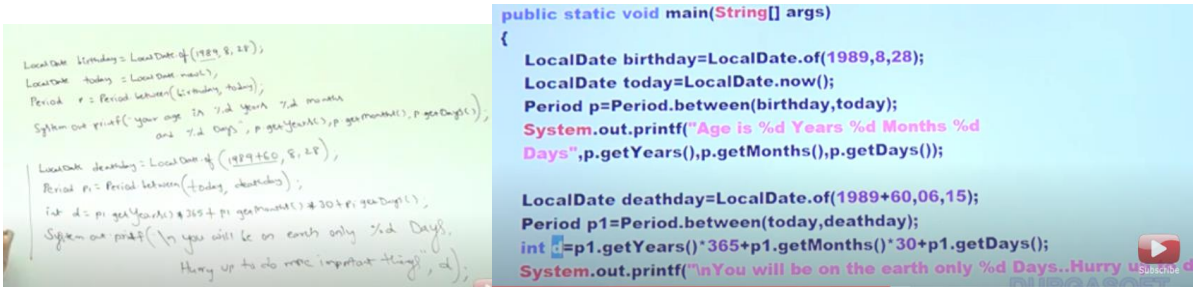
There are may method for date and time, like after 6month or before 6months.




```java
package com.java8.examples;

import java.time.LocalDateTime;
import java.time.Month;

public class Main {

    public static void main(String[] args) {
        // year,month(may be 1,2,3...12), hours,min,secon
        LocalDateTime ld = LocalDateTime.of(2002, Month.MARCH,12,12,12);

        System.out.println(ld);
        System.out.println("After 6th Month :"+ld.plusMonths(6));
        System.out.println("Before 6th Month :"+ld.minusMonths(6));

    }

}
```

```
<terminated> Main [Java Application] C:\Prograr
2002-03-12T12:12
After 6th Month :2002-09-12T12:12
Before 6th Month :2001-09-12T12:12
```

Example-5: ***To find age of a person(period):***

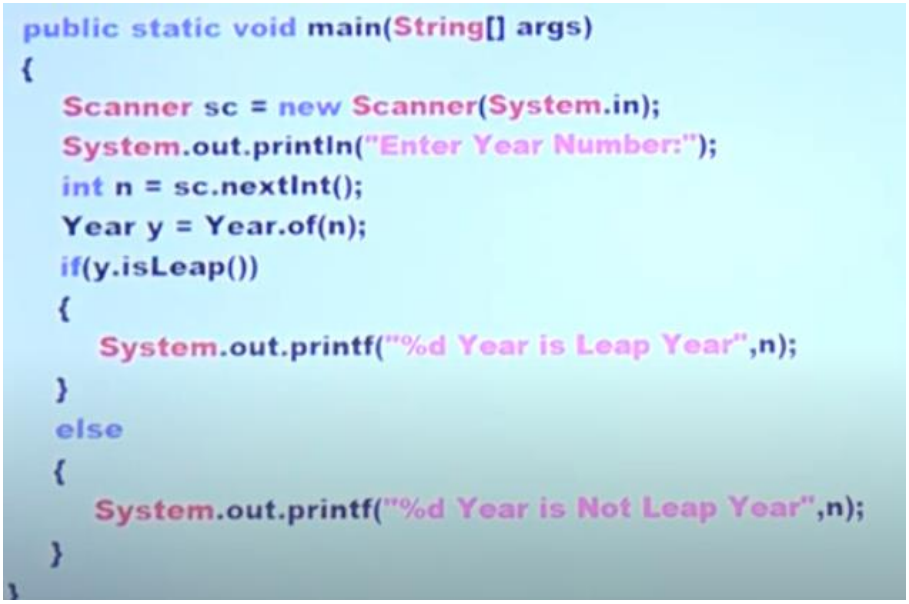


```
public static void main(String[] args)
{
   LocalDate birthday=LocalDate.of(1989,8,28);
   LocalDate today=LocalDate.now();
   Period p=Period.between(birthday,today);
   System.out.printf("Age is %d Years %d Months %d
   Days",p.getYears(),p.getMonths(),p.getDays());

   LocalDate deathday=LocalDate.of(1989+60,06,15);
   Period p1=Period.between(today,deathday);
   int d=p1.getYears()*365+p1.getMonths()*30+p1.getDays();
   System.out.printf("\nYou will be on the earth only %d Days..Hurry u
```

```
1  package com.java8.examples;
2
3  import java.time.LocalDate;
4  import java.time.Period;
5
6  public class Main {
7
8      public static void main(String[] args) {
9
10         LocalDate birthday = LocalDate.of(1990, 10, 21);
11         LocalDate today = LocalDate.now();
12
13         Period period = Period.between(birthday, today);
14
15         System.out.printf("Year: %d, Month: %d, Day: %d", period.getYears(),
16                 period.getMonths(), period.getDays());
17
18     }
19
20 }
```

```
<terminated> Main [Java Application]
Year: 29, Month: 6, Day: 0
```

Example-6: ***To write a program to find given Year is leap year or not by using Year Class:***

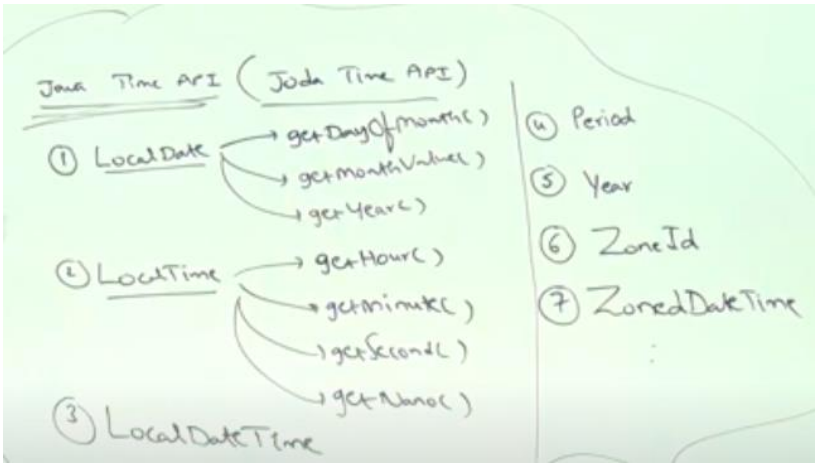


```
public static void main(String[] args)
{
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter Year Number:");
   int n = sc.nextInt();
   Year y = Year.of(n);
   if(y.isLeap())
   {
      System.out.printf("%d Year is Leap Year",n);
   }
   else
   {
      System.out.printf("%d Year is Not Leap Year",n);
   }
}
```

Example-7: ZoneId Class ***– for all over the world date and time example.***

```
public static void main(String[] args)
{
   ZoneId zone=ZoneId.systemDefault();
   System.out.println(zone);

   ZoneId la=ZoneId.of("America/Los_Angeles");
   ZonedDateTime zt=ZonedDateTime.now(la);
   System.out.println(zt);

}
```

```
D:\durga_classes>java Test
Asia/Calcutta
2017-07-15T22:57:58.953-07:00[America/Los_Angeles]

D:\durga_classes>_
```



8.Optional Class:

It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values. It is introduced in Java 8 and is similar to what Optional is in Guava.

Example: Java Program without using Optional

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerExceptio

```
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      String lowercaseString = str[5].toLowerCase();
      System.out.print(lowercaseString);
   }
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at lambdaExample.OptionalExample.main(OptionalExample.java:6)
```

To avoid the abnormal termination, we use Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

Java Optional Example: If Value is not Present

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      Optional<String> checkNull = Optional.ofNullable(str[5]);
      if(checkNull.isPresent()){  // check for value is present or not
         String lowercaseString = str[5].toLowerCase();
         System.out.print(lowercaseString);
      }else
         System.out.println("string value is not present");
   }
}
```

Output:

```
string value is not present
```

Another Java Optional Example

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
      Optional<String> checkNull = Optional.ofNullable(str[5]);
      checkNull.ifPresent(System.out::println);   // printing value by using method reference
      System.out.println(checkNull.get());    // printing value by using get method
      System.out.println(str[5].toLowerCase());
   }
}
```

Output:

```
JAVA OPTIONAL CLASS EXAMPLE
JAVA OPTIONAL CLASS EXAMPLE
java optional class example
```

String.join():

```
Method Syntax

String join(CharSequence delimiter, CharSequence... elements)
```

```
String joinedString = String.join(", ", "How", "To", "Do", "In", "Java");
System.out.println(joinedString);

Output:

How, To, Do, In, Java
```

Convert Stream to List:

```
public class ForEach_Main {

    public static void main(String[] args) {

        Stream<String> stream=Stream.of("Java","Dotnet");

        List<String> list=stream.collect(Collectors.toList());

        list.forEach(System.out::println);

    }
}
```

```
Problems  @ Javadoc  Declaration  Console
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\java
Java
Dotnet
```

Java 8 forEach loop:

The Java forEach is a utility method to iterate over a collection or stream and perform a certain action on each element of it.

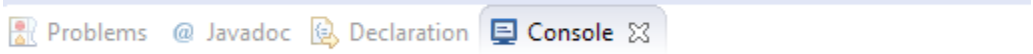
Example 1: List collection-

```
import java.util.function.Consumer;

public class ForEach_Main {

    public static void main(String[] args) {

        List<Integer> list=new ArrayList<>(Arrays.asList(1,2,3,4,5,6));

        Consumer<Integer> c=System.out::println;

        list.forEach(c);

    }
}
```

```
Problems  @ Javadoc  Declaration  Console
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (Feb 19
1
2
3
4
5
6
```

Example 2: Map collection Iteration using java 8 for Each loop.

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

        Map<Integer, Integer> map=new HashMap<>();

        map.put(1, 1);
        map.put(2, 2);
        map.put(3, 3);

        Consumer<Map.Entry<Integer, Integer>> c=System.out::println;

        System.out.println("***** Entry Set *****");
        map.entrySet().forEach(c);

        Consumer<Integer> c1=System.out::println;
        System.out.println("***** Key Set *****");
        map.keySet().forEach(c1);
```




```
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (
***** Entry Set *****
1=1
2=2
3=3
***** Key Set *****
1
2
3
```

```java
public class ForEach_Main {

    public static void main(String[] args) {

        Map<String, Integer> map=new HashMap<>();

        map.put("One", 1);
        map.put("Two", 2);
        map.put("Three", 1);


        Consumer<Map.Entry<String, Integer>> consumer=entry->{
            System.out.println("Key is :"+entry.getKey());
            System.out.println("Value is :"+entry.getValue());
        };

        map.entrySet().forEach(consumer);
```

Problems @ Javadoc Declaration Console

```
<terminated> ForEach_Main [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw
Key is :One
Value is :1
Key is :Two
Value is :2
Key is :Three
Value is :1
```

==Interview Java-8==

1. What are some of the important features that are introduced in Java 8?

```
concise code
Enabling Functional Programming

1. Lambda Expressions
2. Functional Interface
3. Default Methods and static methods
4. Predefined Functional Interfaces
   Predicate
   Function
   Consumer
   Supplier
   etc
5. Double Colon Operator (::)
   Method reference
   Constructor reference
6. Streams
7. Date and Time API
8. Optional class
9. Nashron JavaScript Engine
```

2.What is a lambda expression?

```
It is an anonymous function
     Nameless
     without return type
     without modifiers

Very Very Very Easy!!!!
```

3. What are the most important advantages of using Java 8?

The introduction of Java 8 has been proven to be very helpful for programmers in the following ways:

- Code is now highly readable
- More reusability of code
- Code size is now very compact
- Minimum boilerplate code
- Easier testability methods
- Parallel execution and operations

4. What is the meaning of functional interfaces in Java 8?

Functional interface should contain only one abstract method is called functional interface. It may be or may not be contain non-abstract method but I should contain one Abstract method.

In use annotation(@FunctionalInterface) to declare it's an functional interface

Function Interfaces are like:

```
Runnable===>run()
Callable===>call()
Comparable==>compareTo()
ActionListener==>actionPerformed()
```

Above we have examples.

5. What is the meaning of method reference in Java 8?

Method references are used in Java 8 to refer to methods of functional interfaces. It can be considered as a short-code version of using a lambda expression.

The following is the expression for a method reference:

```
Class::methodname
```

6. Differentiate between a predicate and a function in Java 8.

Predicate	Function
Returns True or False	Returns an object
Used as an assignment target for lambda expressions	Can be used for both lambda expressions and method references

7.What are default methods in Java 8?

Above we have default method details and examples.

8. What are the core API classes for date and time in Java 8?

There are three main core API classes for date and time in Java 8 as given below:

- LocalDate
- LocalTime
- LocalDateTime

9. What were the issues that were fixed with the new Date and Time API of Java 8?

With the older versions of Java, java.util.Date was mutable. This means it has absolutely no thread-safety.

Also, java.text.SimpleDateFormat was not thread-safe in the older versions. In terms of readability as well, the older Date and Time API was difficult to understand for programmers.

10.What is Nashorn in Java 8?

11.What is the use of the optional keyword in Java 8?

12.What is JJS in Java 8?

JJS is the common line tool that comes packaged with Java 8. It is used to run JavaScript code seamlessly using just the console.

13. What are collectors in Java 8?

Collectors are mainly used to combine the final result after the processing of elements in a stream. They are used to return lists or strings.

The following piece of code denotes how collectors work:

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);
```

14.When is an ideal situation to use the Stream API in Java 8?

The Stream API in Java 8 can be effectively used if the Java project calls for the following operations:

1. Perform database operations
2. Execute operations lazily
3. Write functional style programming
4. Perform parallel processing
5. Use pipeline operations
6. Use internal iteration

15. What is a supplier in Java 8?

Above explanation.

16. Differentiate between findFirst() and findAny() in the Stream API of Java 8.

findFirst()	findAny()
Always returns the first element from a stream	Can choose any element present in the stream
Deterministic behavior	Non-deterministic behavior

17. Differentiate between Collection API and Stream API in Java 8.

Collection API	Stream API
Helps store object data	Helps in the computation of data
Stores a limited number of elements	Can store an unlimited number of elements
Eager execution	Lazy execution

Stream API	Collection API
It was introduced in Java 8 Standard Edition version.	It was introduced in Java version 1.2
There is no use of the Iterator and Spliterators.	With the help of forEach, we can use the Iterator and Spliterators to iterate the elements and perform an action on each item or the element.
An infinite number of features can be stored.	A countable number of elements can be stored.
Consumption and Iteration of elements from the Stream object can be done only once.	Consumption and Iteration of elements from the Collection object can be done multiple times.
It is used to compute data.	It is used to store data.

18. What is the meaning of a Spliterator in Java 8?

Spliterator is a newly introduced iterator interface for Java 8. It is very efficient and handles API-related operations seamlessly across the runtime environment.

19. Differentiate between Spliterator and a regular iterator in Java 8.

Spliterator	Iterator
Introduced with Java 8	Present since Java 1.2
Used in Stream API	Used in Collection API
Helps in the iteration of streams in a parallel and sequential order	Iterates collections in a sequential order only
Example: tryAdvance()	Examples: next() and hasNext()

20. Can you name the common types of functional interfaces in the standard library?

There are a lot of functional interface types in the standard library, and some of them are as follows:

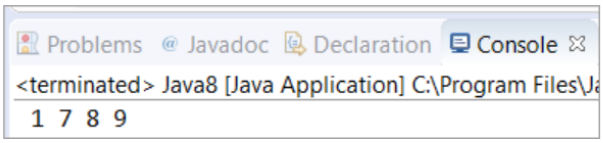
- BiFuction
- BinaryOperator
- Consumer
- Predicate
- Supplier
- UnaryOperator

21. What is the easiest way to find and remove duplicate elements from a list using Java 8?

Duplicate elements can be listed and removed easily by applying stream operations and performing a collection later using the Collections.toSet() method. This should remove all of the duplicate elements present in the list.

```
public class Java8 {
    public static void main(String[] args) {
        Integer[] arr1 = new Integer[] { 1, 9, 8, 7, 7, 8, 9 };
        List<Integer> listdup = Arrays.asList(arr1);

        // Converted the Array of type Integer into List

        Set<Integer> setNoDups = listdup.stream().collect(Collectors.toSet());

        // Converted the List into Stream and collected it to “Set”
        // Set won't allow any duplicates

        setNoDups.forEach((i) -> System.out.print(" " + i));


    }
}
```

Output:



22. What is the easiest way to convert an array into a stream in Java 8?

Any array in Java 8 can be converted into a stream easily using the stream class. The creation of a stream using a factory method is as shown below:

```
String[] testarray = {"Hello", "Intellipaat", "learners"};
Stream numbers = Stream.of(testarray);
numbers.forEach(System.out::println);
```

23. What is a SAM Interface?

Java 8 has introduced the concept of FunctionalInterface that can have only one abstract method. Since these Interfaces specify only one abstract method, they are sometimes called as SAM Interfaces. SAM stands for “Single Abstract Method”.

24. String:: Valueof Expression

It is a static method reference to the ***ValueOf*** method of the **String** class.
System.out::println is a static method reference to println method of out object of System class.
It returns the corresponding string representation of the argument that is passed. The argument can be Character, Integer, Boolean, and so on.

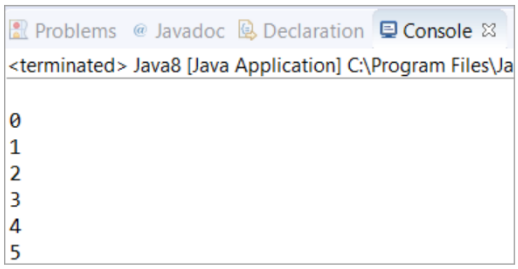
25. What is the difference between limit and skip?

Answer: The limit() method is used to return the Stream of the specified size. **For Example,** If you have mentioned limit(5), then the number of output elements would be 5.

Let's consider the following example. The output here returns six elements as the limit is set to 'six'.

```
import java.util.stream.Stream;

public class Java8 {

    public static void main(String[] args) {
        Stream.of(0,1,2,3,4,5,6,7,8)
        .limit(6)
        /*limit is set to 6, hence it will print the
        numbers starting from 0 to 5
        */
        .forEach(num->System.out.print("\n"+num));
    }
}
```

Output:

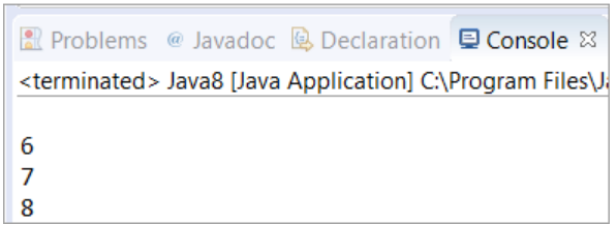


Whereas, the skip() method is used to skip the element.

Let's consider the following example. In the output, the elements are 6, 7, 8 which means it has skipped the elements till the 6th index (starting from 1).

```
import java.util.stream.Stream;

public class Java8 {

    public static void main(String[] args) {
        Stream.of(0,1,2,3,4,5,6,7,8)
        .skip(6)
        /*
         It will skip till 6th index. Hence 7th, 8th and 9th
         index elements will be printed
         */
        .forEach(num->System.out.print("\n"+num));
    }
}
```

Output:



26. Write a program to print 5 random numbers using forEach in Java 8?

Answer: The below program generates 5 random numbers with the help of forEach in Java 8. You can set the limit variable to any number depending on how many random numbers you want to generate.

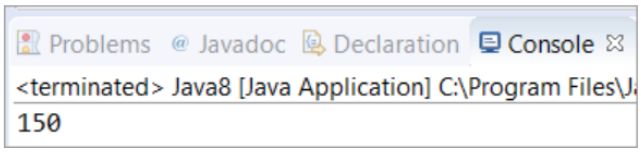
```
import java.util.Random;

class Java8 {
    public static void main(String[] args) {

        Random random = new Random();
        random.ints().limit(5).forEach(System.out::println);
        /* limit is set to 5 which means only 5 numbers will be printed
        with the help of terminal operation forEach
        */

    }
}
```

Output:

27. Write a Java 8 program to get the sum of all numbers present in a list?

```
import java.util.*;

class Java8 {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();

        list.add(10);
        list.add(20);
        list.add(30);
        list.add(40);
        list.add(50);
        // Added the numbers into Arraylist
        System.out.println(sum(list));
    }

    public static int sum(ArrayList<Integer> list) {
        return list.stream().mapToInt(i -> i).sum();
        // Found the total using sum() method after
        // converting it into Stream
    }
}
```

Output:



28.Write a program - Given a list of employees, you need to filter all the employee whose age is greater than 20 and print the employee names.(Java 8 APIs only)

```
List<String> employeeFilteredList = employeeList.stream()
                    .filter(e->e.getAge()>20)
                    .map(Employee::getName)
                    .collect(Collectors.toList());
```

29. Given the list of employees, find the employee with name “Mary”.

```
List<Employee> employeeList = createEmployeeList();
        Optional<Employee> e1 = employeeList.stream()
                    .filter(e->e.getName().equalsIgnoreCase("Mary")).findAny();

        if(e1.isPresent())
            System.out.println(e1.get());
```