Chapter 1 : Introduction To Java

History of Java

- Java was developed by James Gosling at Sun Microsystems, Inc. (Started in 1988).
- Java Originated at Sun Microsystems, Inc. 1991.
- On January 27, 2010, Sun Microsystems was acquired by Oracle corporation for 7.4 billion US Dollars.

Java is Everywhere

- Java resides in Mobile, Client Machines, Server Machines, Embedded Devices, Smart Phones, etc.
- It shared the same basic features of the language and libraries.
- Principle of Java: “Write Once, Run Anywhere” (WORA) or sometimes “Write Once, Run Everywhere” (WORE).

Java Flavors

- J2SE (Java Standard Edition)
- JEE (Java Enterprise Edition)
- JME (Java Micro Edition for Mobiles) and so on...

Why Java?

- Java is platforms independent (Windows, Mac, Linux, Raspberry Pi, etc).
- It is high level programming language.
- It is one of the most popular programming language in the world.
- It is easy to learn and simple to use.
- It is open-source and free.
- It is secure, fast and powerful.
- It has huge community support (Millions of Developers).
- Java is an Object Oriented Programming Language which gives a clear structure to programs and allows code to be reused, lowering development costs.

What is Java Library?

- Java Library is a collection of predefined classes.
- You can use these classes either by inheriting them or by instantiating them.

Features of Java

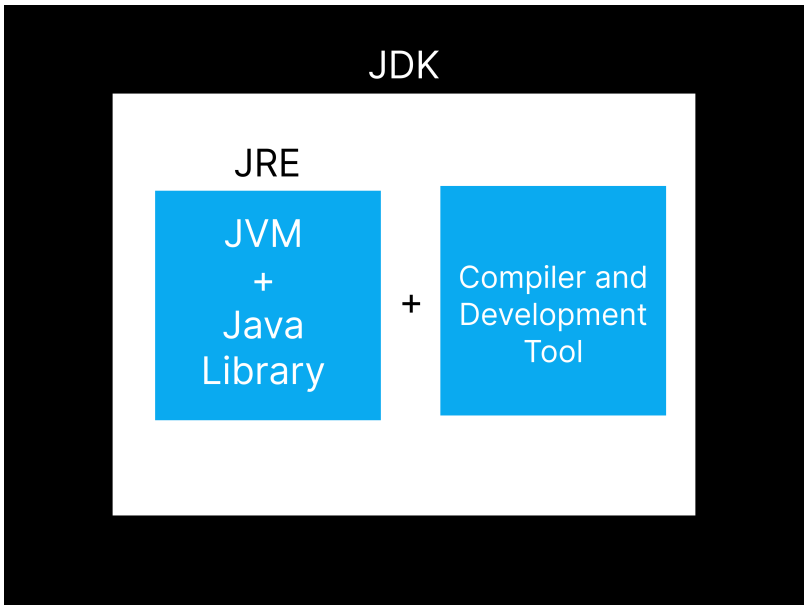
- Simple
- Object Oriented
- Distributed
- Interpreted
- Robust
- Secure
- Portable
- Multi-Threaded
- Garbage Collector

Points to Remember

- Java is a case sensitive language like C and C++.
- Java is nearly 100% object oriented language (means not totally object oriented). In Java, it is not possible to make a function which is not a member of any class (as we can do in C++).
- Java language is totally made without Pointers (In C and C++ we are worked on pointer).

JDK (Java Development Kit)

- It is a development kit which includes all predefined classes.
- It contains JRE and JVM.



JRE (Java Runtime Environment)

- It is usd to successfully execute the ".class" file i.e. bytecode

JVM (Java Virtual Machine)

- The execution of bytecode means that ".class" file is done by the JVM.
- JVM has 3 modules :

Java Program

```
public class MyFirstProgram{
    public statioc void main( String args[]){
        System.out.println("Hello World!");
    }
}
```

OUTPUT:
Hello World!

Chapter 2 : Java Keywords

Keywords	Keywords	Keywords	Keywords	Keywords
abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	enum	import	public	transient
byte	else	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	-
continue	for	null	synchronized	-

Chapter 3 : Java Data Types

- A type identifies a set of values (and their representation in memory) and a set of operations that transform these values into other values of that set.
- Java is strongly typed language (means every variable must be with a data type).

There are two types of data type:
1. Primitive Data Types : Predefined or Build-in data types.

Data Type	Keyword	Size(Byte)	Size(Bit)
character	char	2	16
byte	byte	1	8
short	short	2	16
integer	int	4	32
float	float	4	32
long	long	8	64
double	double	8	64
boolean	boolean	NA	NA

2. User-Defined or Reference Data Types : Data types which are created by the users.

Note:

- Primitive types has always a value but reerence types has no value i.e. null.
- Primitive data types always starts with a **Lowercase** while Reference data types starts with **Uppercase**.

Chapter 4 : Java Variables

Java Variables are declared as follows:

```
data_type variable_name = data;
```

Example

```
int num = 10;
char alphabet = "a";
```

Chapter 5 : Java Comments

Single Line Comments -

```
//this is a single line comment.
```

Multi Line Comments -

```
/*
this is multi line comments.
it will comment a group of lines.
*/
```

Chapter 6 : Java Constants

- Integer constants consists of a sequesnce of digits.
- If the constant is to represent a long integer value, it must be suffixed with an uppercase L or lowercase l.
- If there is no suffix the constant represents 32 bit integer (i.e. an int).
- Integer constant can be specified in decimal, hexadecimal, octal or binary format.

Example

```
127
0x7f
0177
0101101100
```

Chapter 7 : Java Type Casting

Type casting we have to use when we assign a value of one primitive data type to another type.
There are two types of casting:
1. Widening Casting: Converting a smaller type to a larger type.

```
byte -> short -> char -> int -> long -> float -> double
```

Example

```
public class Example{
    public static void main (String args[]){
        int num = 9;
        double doubleValue = num;
        System.out.println(num);
        System.out.println(doubleValue);
    }
}
```

OUTPUT:
9
9.0

2. Narrowing Casting: Converting a larger type to a smaller type.

```
double -> float -> long -> int -> char -> short -> byte
```

Example

```
public class Example{
    public static void main (String args[]){
        double bill = 9.78;
        int num = (int)bill;
        System.out.println(bill);
        System.out.println(num);
    }
}
```

OUTPUT:
9.78
9

Chapter 8 : Java Operators

There are five types of operators in Java:
1. Arithmetic Operator:

- Addition
- Subtraction

- Multiplication
- Division
- Modulus
- Increment
    1. pre-increment
    2. post-increment
- Decrement
    1. pre-decrement
    2. post-decrement

Example

```
public class ArithmeticOperations{
    public static void main (String args[]){
        int a = 10;
        int b = 20;
        System.out.println("a+b = "+(a+b));
        System.out.println("a-b = "+(a-b));
        System.out.println("a*b = "+(a*b));
        System.out.println("a/b = "+(a/b));
        System.out.println("a%b = "+(a%b));
    }
}
```

OUTPUT:
a+b = 30
a-b = -10
a*b = 200
a/b = 2
a%b = 0

2. Comparison Operators:

Operators	Use
==	Equal To
!=	Not Equal To
>	Greater Than
<	Less than
>=	Greater Than Equal To
<=	Less Than Equal To

3. Logical Operators:

Operators	Use
&& (Logical AND)	Returns true if both the conditions are true
|| (Logical OR)	Returns true if one of the condition is true
! (Logical NOT)	Reverse the result, returns false if the result is true

4. Assignment Operators:

Operators	Example	Same as
=	x=5	x=5
+=	x+=3	x=x+3
-=	x-=3	x=x+3
*=	x*=3	x=x+3
/=	x/=3	x=x+3
%=	x%=3	x=x+3
&=	x&=3	x=x+3
|=	x|=3	x=x+3
^=	x^=3	x=x+3
>>=	x>>=3	x=x+3
<<=	x<<=3	x=x+3

5. Bitwise Operators:

Operators	Use
& (AND)	Set each bit to 1 if both bits are 1
| (OR)	Sets each bit to 1 if any of the two bits is 1
~ (NOT)	Invert all the bits (1's Compliment)
^ (XOR)	Sets each bit to 1 if only one of the two bits is 1
<< (Zero-fill left shift)	Shift left by pushing zeroes in from the right side and letting the almost bits fall off
>>> (Zero-fill right shift)	Shift right by pushing zeroes in from the left and letting the rightmost bits dall off
>> (Signed right shift)	Shift right by pushing copies of the leftmost bit in from the left and letting the rightmost bit fall off

Chapter 9 : Java Conditional Statements

Here we use the two keywords to implements the conditional statements seperately or we can combine them to create a nested conditional statements.

Example of conditional statements

```
int i = 10;

if(i>5){
    System.out.println("Greater than 5");
}
else{
    System.out.println("Smaller than 5");
}
```

OUTPUT:
Greater than 5

Example of nested conditional statements

```
int i = 10;

if(i>5){
    System.out.println("Greater than 5");
}
else if(i==0){
    System.out.println("Zero Value");
}
else{
    System.out.println("Smaller than 5");
}
```

OUTPUT:
Greater than 5

Note

- You can use only **if** keyword only.
- You can also use **if** with **else if** keyword.
- But you can not write **else if** or **else** directly.

Ternary Operator:

- It is related to if else conditions but in unique way.

```
Syntax : variable = condition ? expression1 : expression2
```

Example

```
public class Example{
    public static void main (String args[]){
        int i = 8;
        int j = 0;
        System.out.println(j=i>6?1:2);
    }
}
```

OUTPUT:
1

Chapter 10 : Java Break/Continue

- break and continue keywords are used with loops.

1. break : break keyword can be used to jump out of a loop.

Example

```
public class Example{
    public static void main(String args[]){
        for(int i=0;i<10;i++){
            if(i==4){
                break;
            }
            System.out.println(i);
        }
    }
}
```

OUTPUT:
0
1
2
3

2. continue : continue statements skip a specific value and continue the loop.

Example

```
public class Example{
    public static void main(String args[]){
        for(int i=0;i<10;i++){
            if(i==4){
                continue;
            }
            System.out.println(i);
        }
    }
}
```

OUTPUT:
0
1
2
3
5
6
7
8
9

Chapter 11 : Java Loops

We have mainly three types of loop:
1. while loop:

Example

```
public class Example{
    public static void main (String args[]){
        int i=0;
        while(i<5){
            System.out.println(i);
            i++;
        }
    }
}
```

OUTPUT:
0
1
2
3
4

2. do/while loop:

Example

```
public class Example{
    public static void main(String args[]){
        int i=0;
        do{
            System.out.println(i);
            i++;
        }
        while(i<5);
    }
}
```

```
OUTPUT:
0
1
2
3
4
```

3. for loop:

Example

```
public class Example{
    public static void main(String args){
        for(int i=0;i<5;i++){
            System.out.println(i);
        }
    }
}
```

```
OUTPUT:
0
1
2
3
4
```

Note

- We can also use for loop as a **for each loop** as :

Example

```
for(item:array_list){
    //some statements
}
```

Chapter 12 : Java Switch Statements

Example

```
public class Example{
    public static void main (String args[]){
        int day = 4;
        switch(day){
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            case 4:
                System.out.println("Thursday");
                break;
            case 5:
                System.out.println("Friday");
                break;
            case 6:
                System.out.println("Saturday");
                break;
            case 7:
                System.out.println("Sunday");
                break;
        }
    }
}
```

OUTPUT:
Thursday

> If we have to add any default case then we can add default case at the bottom of the switch cases. make sure there is no need of break in default case.

Chapter 13 : Java Methods

- A method is a block of code which only runs when it is called.
- You can pass data, known as parameters into a method.
- Methods are used to perform certain actions and they are also known as functions.
- Methods once we declared we can call them multiple times as much as we want.

Why we use methods?

To reuse the code, defines the code once and use it many times.

Example

```
public class Example{

    //created static method
    static void myMethod(){
        System.out.println("Hey Programmers");
    }

    public static void main (String args[]){
        myMethod(); //calling method here
    }
}
```

OUTPUT:
Hey Programmers

Java Methods Parameters

- Information can be passed to methods as **parameter**.
- Parameters act as variables inside the method.
- Parameters are specified after the method name inside the parenthesis. You can ad as many parameters as you want, just separate them with **comma**.

Example

```
public class Example{

    //created static method
    static void myMethod(String str){
        System.out.println(str);
    }

    public static void main (String args[]){
        String name = "Rajat";
        myMethod(name); //calling method here
    }
}
```

OUTPUT:
Rajat

- In above example, **name** has been passed as an **argument** and at method declaration **str** which we have passed it is a **parameter**.
- We can pass multiple paramters as per our need.

Chapter 15 : Java Strings

Immutable	Mutable
String s = new String("Rajat"); s.concat("Kotangale"); System.out.println(s); **OUTPUT:** Rajat	StringBuffer s = new StringBuffer("Rajat"); s.append("Kotangale"); System.out.println(s); **OUTPUT:** RajatKotangale
Once we created a string object, we can't perform any changes in the existing object. If we are trying to perform any changes with string then **a new object will be created**. This behaviour nothing but a Immutable nature of String object.	Once we created a StringBuffer object, we can perform any changes on the screen it will be **modified on the same object**. This behaviour is nothing but a mutable nature of StringBuffer object.
s ——-> Rajat =>No changes RajatKotangale => new object created after concatenation. but the original string will remain same as it is i.e. **"Rajat"**	s ——->Rajat s —-> RajatKotangale => Here new object is not created. **"Rajat"** object is now modified as **"RajatKotangale"**

String	StringBuffer
In string class .equals() method is overridden for **content comparison**. Hence .equals() method returns true if content is same eventhough object is different.	In StringBuffer class .equals() is not overridden. Hence it check and compare the **reference of both object**, due to this it return false, even if content is same.
String s = new String("Rajat"); In this case two object will be created, one is in the **Heap Area** and another one is in the **String Pool Constant(SCP)** of the memory. But if we try to make any modification then a new object will be created in both of the area's of memory.	StringBuffer s = new StringBuffer("Rajat"); In this case two objects will be created, one is in the **Heap Area** and another one is in the **String Pool Constant** of the memory. But if we try to make any modification then the heap area reference object remains same as it is changes only happens in the string pool constant.
Heap Area | String Pool Constant Rajat | Rajat RajatKotangale | RajatKotangale	
String str = "Rajat"; In this case only one object is created in String Pool Constant str is point to the object **Heap Area | String Pool Constant** | Rajat	

- Garbage Collector is not allowed in SCP Area

String Pool Constant

- Object creation in SCP is always optional. First JVM checks there is any object already present with similar content, if it is available then it will reuse existing object instead of creating new object.
- But, If the similar content object not available then new object is created. So, duplicate object is not allowed in SCP.

Example

```
String s1 = new String("Rajat");
String s2 = new String("Rajat");
String s3 = "Rajat";
String s4 = "Rajat";
```

Heap	String Pool Constant
s1 ——-> Rajat	Rajat => s3 and s4 directly use this object from string pool constant
s2 ——-> Rajat	

```
String s1 = new String("Rajat");
s1.concat("Kotangale");
s1.concat("Solution");
```

Heap	String Pool Constant
s1 ——-> Rajat	Rajat
RajatKotangale	Kotangale
RajatSolution	Solution

String

- A string is an object representing a sequence of characters.
- It's a fundamental data type and is part of the java.lang package, which means it's readily available in all Java programs without needing to import any additional packages.
- Strings in Java are immutable, meaning their values cannot be changed after they are created.
- This immutability ensures that once a string object is created, its contents remain constant throughout its lifetime.
- Strings in Java are typically enclosed in double quotes ("").

Methods:

1. charAt(int index) - Returns the character at the specified index.
2. concat(String str) - Concatenates the specified string to the end of this string.
3. contains(CharSequence s) - Returns true if this string contains the specified

sequence of characters.
4. endsWith(String suffix) - Tests if this string ends with the specified suffix.
5. equals(Object anObject) - Compares this string to the specified object.
6. equalsIgnoreCase(String anotherString) - Compares this string to another string, ignoring case considerations.
7. indexOf(int ch) - Returns the index of the first occurrence of the specified character.
8. indexOf(String str) - Returns the index of the first occurrence of the specified substring.
9. isEmpty() - Returns true if, and only if, length() is 0.
10. length() - Returns the length of this string.
11. replace(char oldChar, char newChar) - Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
12. replace(CharSequence target, CharSequence replacement) - Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
13. split(String regex) - Splits this string around matches of the given regular expression.
14. startsWith(String prefix) - Tests if this string starts with the specified prefix.
15. substring(int beginIndex) - Returns a new string that is a substring of this string.
16. substring(int beginIndex, int endIndex) - Returns a new string that is a substring of this string.
17. toLowerCase() - Converts all of the characters in this String to lower case using the rules of the default locale.
18. toUpperCase() - Converts all of the characters in this String to upper case using the rules of the default locale.
19. trim() - Returns a copy of the string, with leading and trailing whitespace omitted.
20. valueOf(int i) - Returns the string representation of the int argument.

StringBuffer

- A StringBuffer is a class in Java that provides a mutable sequence of characters.
- It is similar to the String class, but unlike strings, StringBuffers can be modified once they are created.
- This makes StringBuffer useful for situations where you need to manipulate strings frequently without creating new objects each time.
- StringBuffer provides methods for appending, inserting, deleting, and modifying characters in the sequence.
- It is often used in situations where efficient string manipulation is required, such as when building long strings dynamically.

Methods:

1. append() - Appends the specified string representation to the end of the StringBuffer.
2. capacity() - Returns the current capacity of the StringBuffer.
3. charAt(int index) - Returns the character at the specified index in the StringBuffer.
4. delete(int start, int end) - Removes the characters from the specified start index to the end index - 1.
5. deleteCharAt(int index) - Removes the character at the specified index.
6. ensureCapacity(int minCapacity) - Ensures that the capacity of the StringBuffer is at least equal to the specified minimum capacity.
7. getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) - Copies characters from this StringBuffer into the destination character array.
8. indexOf(String str) - Returns the index within this StringBuffer of the first occurrence of the specified substring.
9. insert(int offset, String str) - Inserts the specified string into the StringBuffer at the specified position.
10. insert(int offset, char c) - Inserts the specified character into the StringBuffer at the specified position.
11. length() - Returns the length (number of characters) of the StringBuffer.
12. replace(int start, int end, String str) - Replaces the characters from the start index to the end index - 1 with the specified string.
13. reverse() - Reverses the characters in the StringBuffer.
14. setCharAt(int index, char ch) - Sets the character at the specified index to the specified character.
15. setLength(int newLength) - Sets the length of the StringBuffer to the specified length.
16. substring(int start) - Returns a new String that contains a subsequence of characters currently contained in the StringBuffer, starting from the specified index.
17. substring(int start, int end) - Returns a new String that contains a subsequence of characters currently contained in the StringBuffer, starting from the specified start index and ending at the specified end index - 1.
18. toString() - Returns a string representing the data in the StringBuffer.

StringBuilder

- StringBuilder is a class that provides a mutable sequence of characters, similar to StringBuffer but with some differences in synchronization.
- It is part of the java.lang package and is commonly used for efficient string manipulation when thread safety is not a concern.

Methods

1. append(String str): Appends the specified string to the end of the StringBuilder.
2. append(Object obj): Appends the string representation of the specified object to the end.
3. append(char c): Appends the specified character to the end.
4. append(CharSequence s): Appends the specified character sequence to the end.
5. insert(int offset, String str): Inserts the specified string at the specified position.
6. insert(int offset, Object obj): Inserts the string representation of the specified object at the specified position.
7. insert(int offset, char c): Inserts the specified character at the specified position.
8. insert(int offset, CharSequence s): Inserts the specified character sequence at the specified position.
9. delete(int start, int end): Deletes the characters from the start index to the end index - 1.
10. deleteCharAt(int index): Deletes the character at the specified index.
11. replace(int start, int end, String str): Replaces the characters from the start index to the end index - 1 with the specified string.
12. substring(int start): Returns a new StringBuilder that contains a subsequence of characters currently contained in the StringBuilder, starting from the specified index.
13. substring(int start, int end): Returns a new StringBuilder that contains a subsequence of characters currently contained in the StringBuilder, starting from the specified start index and ending at the specified end index - 1.
14. reverse(): Reverses the order of characters in the StringBuilder.
15. length(): Returns the length (number of characters) of the StringBuilder.
16. setCharAt(int index, char ch): Sets the character at the specified index to the specified character.
17. charAt(int index): Returns the character at the specified index.
18. indexOf(String str): Returns the index within the StringBuilder of the first occurrence of the specified substring.
19. lastIndexOf(String str): Returns the index within the StringBuilder of the last occurrence of the specified substring.
20. toString(): Returns a string representing the data in the StringBuilder.

Chapter 16 : Java Maths

- It allow us to perform mathematical tasks on numbers.
- Math is a class in java where so many methods are present to perform those tasks.

Examples

1. **Math.max(x,y)** : It is used to find highest value between x and y.
2. **Math.min(x,y)** : It is used to find minimum value etween x and y.
3. **Math.sqrt(x)** : It is used to find square root.
4. **Math.abs(x)** : It is used return absolute positive value.
5. **Math.random()** : It is used to return a random value between 0.0(inclusive) and 1.0(exclusive).

and so on...

Chapter 17 : Java Command-Line Arguments

- A Java application can accept any number of arguments from the command line.

Example 1

```
public class Example{
    public static void main(String args[]){
        System.out.println(args[0]);
    }
}
```

OUTPUT: Command Line Execution
javac Example.java
java Example Rajat
Rajat

Example 2

```
public class Example{
    public static void main(String args[]){
        for(int i=0; i<args.length;i++){
            System.out.println(args[0]);
        }
    }
}
```

OUTPUT: Command Line Execution
javac Example.java
java Example Rajat Sanjiv Kotangale
Rajat
Sanjiv
Kotangale

Chapter 18 : Java Classes

Before learn about classes we need to understand the term object oriented programming.

OOP (Object Oriented Programming)

- It is a programming procedure which consists of objects and objects are nothing but an instance of class which has properties and behavior.

Class

- Using class keyword we implement the encapsulation.
- Encapsulation is a process of binding data and code into a single unit. This is called Encapsulation.
- Class is a description of an object's property and behavior.
- Creating a class is as good as creating data type
- Class is defining a category of data.
- It is a logical entity.
- It is declared using **class** keyword
- Class can be public, default, abstract and final only. We cannot use static, private keywords in outer class.
- Class cannot be declared public more than 1 times in a single file.

Example

```
1  public class Example{
2
3  }
```

Chapter 19 : Java Objects

- Object is a real world entity.
- Object is an instance of a class.
- Object consumes memory to hold property values.

Example

```
1  package java_tutorials_by_rajat;
2
3  class Box{
4      private int length, breadth, height;
5
6      public void setDimension(int l, int b, int h){
7          this.length = l;
8          this.breadth = b;
9          this.height = h;
10     }
11
12     public void getDimention(){
13         System.out.println("Length : "+ length);
14         System.out.println("Breadth : "+ breadth);
15         System.out.println("Height : "+ height);
16     }
17 }
18
19 public class ObjectExample {
20     public static void main(String[] args) {
21         Box b1 = new Box();
22         b1.setDimension(10, 20, 30);
23         b1.getDimention();
24     }
25 }
```

OUTPUT:
Length : 10
Breadth : 20
Height : 30

If we assign b1 new instance of box then create a new object to the same reference therefore, the object we have created is destroyed and the values that we have assigned before are destroyed and new values are assigned to it as 0, 0, 0.

Example

```
package java_tutorials_by_rajat;

class Box{
    private int length, breadth, height;

    public void setDimension(int l, int b, int h){
        this.length = l;
        this.breadth = b;
        this.height = h;
    }

    public void getDimention(){
        System.out.println("Length : "+ length);
        System.out.println("Breadth : "+ breadth);
        System.out.println("Height : "+ height);
    }
}

public class ObjectExample {
    public static void main(String[] args) {
        Box b1 = new Box();
        b1.setDimension(10, 20, 30);
        b1.getDimention();
        b1 = new Box();
        b1.getDimention();
    }
}
```

OUTPUT:
Length : 10
Breadth : 20
Height : 30
Length : 0
Breadth : 0
Height : 0

Chapter 20 : Java Wrapper Classes

- Java is 99% Object-Oriented (Due to primitive data types. Primitive data types are not object types).
- Java is an object-oriented language and as said everything in java is an object.

- Primitive or Reference data types are sort left out in the world of objects that is, they cannot participate in the object activities.
- Therefore, we use wrapper class to fulfill this fault to make java as fully object-oriented.
- So basically, we are going to create objects of different data types using wrapper class.
- We can also convert them from primitive to object and object to primitive.

There are wrapper class for every primitive data type in java:

Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Char
short	Short
int	Integer
long	Long
float	Float
double	Double

This is applicable till version number 1.4 of java

Unboxing	Boxing
Primitive ——————> Objects Primitives are converted into objects	Primitive ——————> Objects Primitives are converted into objects
xxxValue()	valueOf()

Example of Boxing

```
package java_tutorials_by_rajat;

public class WrapperClassExample {
    public static void main(String[] args) {
        int i = 12;
        float j = 3.14f;
        Integer i1 = Integer.valueOf(i);
        Float f1 = Float.valueOf(j);
        System.out.println(i1);
        System.out.println(f1);
    }
}
```

OUTPUT:
12
3.14

Example of Unboxing

```
package java_tutorials_by_rajat;

public class WrapperClassExample {
    public static void main(String[] args) {
        Integer i1 = new Integer(12);
        Float f1 = new Float(3.14f);
        int i = i1.intValue();
        float j = f1.floatValue();
        System.out.println(i);
        System.out.println(j);
    }
}
```

OUTPUT:
12
3.14

From version number 1.5 to above it is converted to :

Auto-Unboxing	Auto-Boxing
Primitive ——————> Objects Primitives are converted into objects	Primitive ——————> Objects Primitives are converted into objects

In this scenario we can pass directly primitives into objects and objects into primitives. Everything is managed by java automatically.

Example of Auto Boxing

```
package java_tutorials_by_rajat;

public class WrapperClassExample {
    public static void main(String[] args) {
        int i = 12;
        float j = 3.14f;
        Integer i1 = i;
        Float f1 = j;
        System.out.println(i1);
        System.out.println(f1);
    }
}
```

OUTPUT:
12
3.14

Example of Auto Unboxing

```
package java_tutorials_by_rajat;

public class WrapperClassExample {
    public static void main(String[] args) {
        Integer i1 = 12;
        Float f1 = 3.14f;
        int i = i1;
        float j = f1;
        System.out.println(i);
        System.out.println(j);
    }
}
```

OUTPUT:
12
3.14

Methods:

1. valueOf():

- Static method (if method is static then there is no need to create an object).
- Return object reference of relative wrapper class.

2. parseXXX():

- Static method
- XXX can be replaced by any primitive data type
- It returns XXX types values.

3. xxxValue():

- Instance method of wrapper class.
- XXX can be replaced by any primitve data type.
- Returns corresponding primitive type.

Chapter 21 : Java Inner Classes/Nested Classes

- Classes inside classes
- We can declare inner class as default, public, abstract, final, static, private and protected.

There are mainly two types:

1. Non Static

- We can not make static function in non static inner class. But we can make static final function.
  In Non Static we have three types:
  A)Instance Inner Class:
  B)Local Inner Class
  C)Anonymous Inner Class

2. Static

- We can access those classes directly because they are static.
- We can also add main method within the inner class it will work normally.

Instance Inner Class

Example

```
1  package com.rajatkotangale;
2
3  public class InstanceInnerClassExample {
4      class Demo{
5          private int i;
6
7          public void setInteger(int num){
8              this.i=num;
9          }
10
11         public int getInteger(){
12             return i;
13         }
14     }
15
16     public static void main(String[] args) {
17         InstanceInnerClassExample i1 = new InstanceInnerClassExample();
18         InstanceInnerClassExample.Demo d1= i1.new Demo();
19         d1.setInteger(10);
20         System.out.println(d1.getInteger());
21     }
22 }
```

OUTPUT:
10

Local Inner Class

Example

```
package com.rajatkotangale;

public class LocalInnerExample {
    public void printSomething(){
        class Demo{
            //created constructor, we can also make methods.
            public Demo(){
                System.out.println("Working...");
            }
        }
        /*make sure to create object below the inner class
        at the time of local inner class*/
        Demo d1 = new Demo();
    }
    public static void main(String[] args) {
        LocalInnerExample li = new LocalInnerExample();
        li.printSomething();
    }
}
```

OUTPUT:
Working...

Anonymous Inner Class

Example

File 1:

```
package com.rajatkotangale;

public class Demo {
    abstract class A{
        protected int i;
        abstract void setInteger();
        public int getInteger(){
            return i;
        }
    }
}
```

File 2:

```
package com.rajatkotangale;

public class AnonymousClassExample {
    public static void main(String[] args) {
        Demo d1 = new Demo();
        Demo.A a= d1.new A(){
            public void setInteger(){
                this.i=10;
            }
        };
        a.setInteger();
        System.out.println(a.getInteger());
    }
}
```

OUTPUT:
10

Static Inner Class

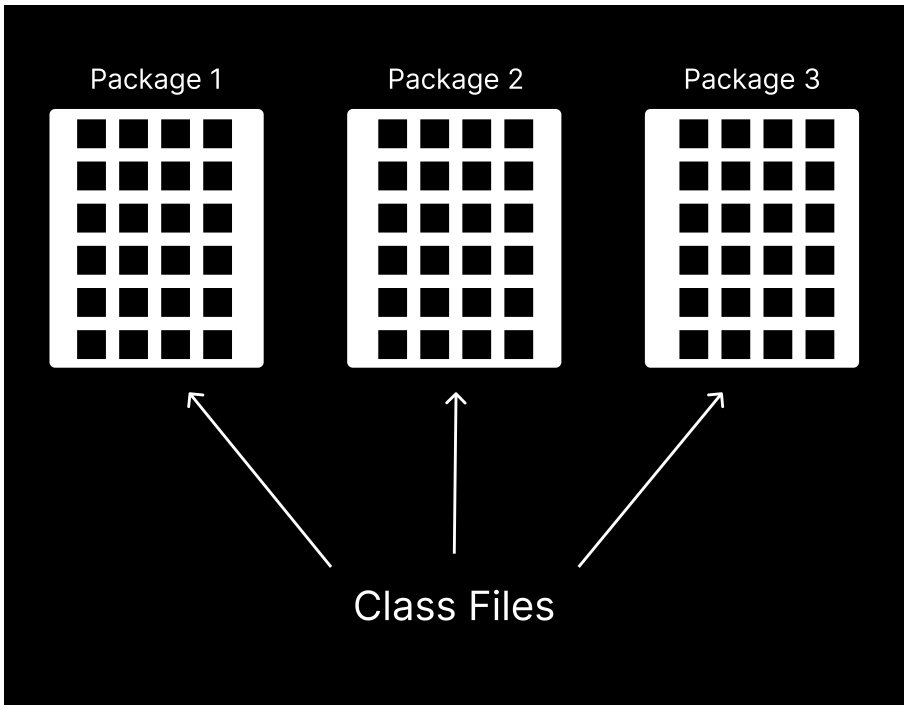
- We can use as normal without creating object we can access them.
- If we created a inner class static then we can also run main method in it.

Chapter 22 : Java Packages

- Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong too.
- Files in one directory (or package) would have different functionality from those of another directory.

Example

> Files in java.io (http://java.io) do something related to I/O, But files in java.net (http://java.net) package give us the way to deal with the network.



Advantages of package

- packaging also help us to avoid class name collision when we use the same class name as that of other.
- The benefits of using package reflet the ease of maintenance, organization, and increase collaboration among developers.

Note: We can create same class name file in different package but we can not create same name class file in same package.

Process to create java package following steps are required :

```
package world;

public class example{
    public static void main (String args[]){
        System.out.println("Hello World");
    }
}
```

Step 1: Open terminal.
Step 2: Go to the file directory using commands.
Step 3: Then compiler the java file and create the package by using following command:

```
javac -d . filename.java
```

Step 4: Run the java program by command

```
java package_name.filename
```

```
OUTPUT:
cd Documents
cd Java Programs
javac -d . Example.java
java.world.Example
Hello World
```

Points to Remember

- We can have only **one public class** in single java file.
- Name of the file should be same as the name of public class.
- In absence of public class, any class name can be given to the file name.
- Only public class can be accessed directly from outside of the package.

Chapter 23 : Java Access Modifiers

Java supports four categories of accessibility rules:

1. private : When members of the class are private, they can not accessed from outside of the class body. They are meant to be **accessed from the same class** in which they are declared.

2. protected : When members are protected, they can be accessed from **any class of the same package** and if we want to use in diffrent package, we have to extends the class that we want to use.

3. public : When members are public, they are accessible from **any class of any package**.

4. default : When members are default, they are accessible **only from the any class of same package**.

Example

```
//public class
public class Example{
    //private variable
    private int x;

    //public function
    public void setValue(int num){
        x = num;
    }

    //public function
    public int getValue(){
        return x;
    }

    public static void main(String args[]){
        Example example = new Example();
        example.setValue(10);
        System.out.println(example.getValue());
    }
}
```

OUTPUT:
10

Note:

- Modifiers can be used for class, member variables and member functions.
- Outer class can be public or default only.
- But in inner class we can use all four access modifiers.

Chapter 24 : Java Inheritance

- Inheritance is the feature by using it child class can get all properties and behaviour of the parent class.

Syntax:

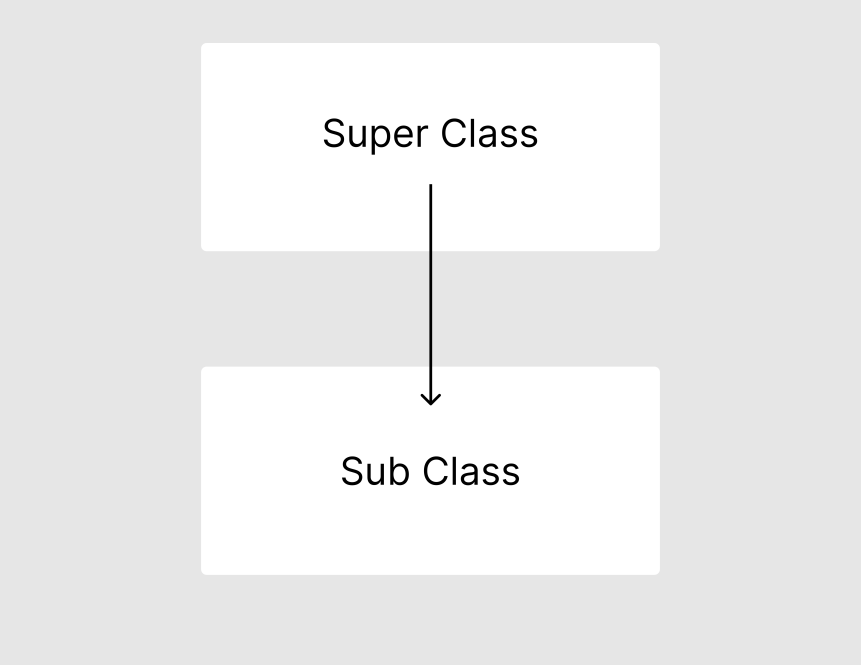
```
class SubClass extends SuperClass{

}
```

- **extends** is a keyword.
- Base class means superclass(parent).
- Derived class means subclass(child).

Types of Inheritance in Java

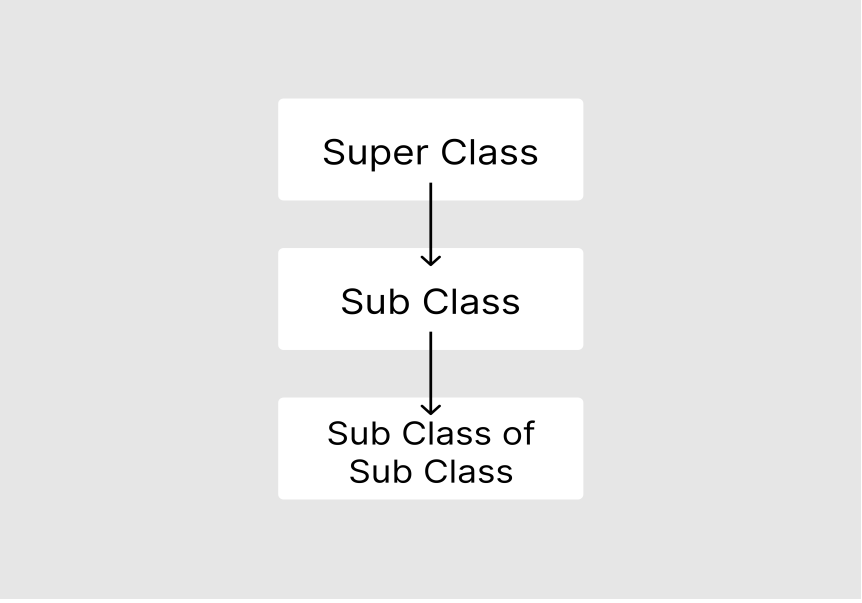
1. Single Inheritance:

- one superclass and it's one subclass.



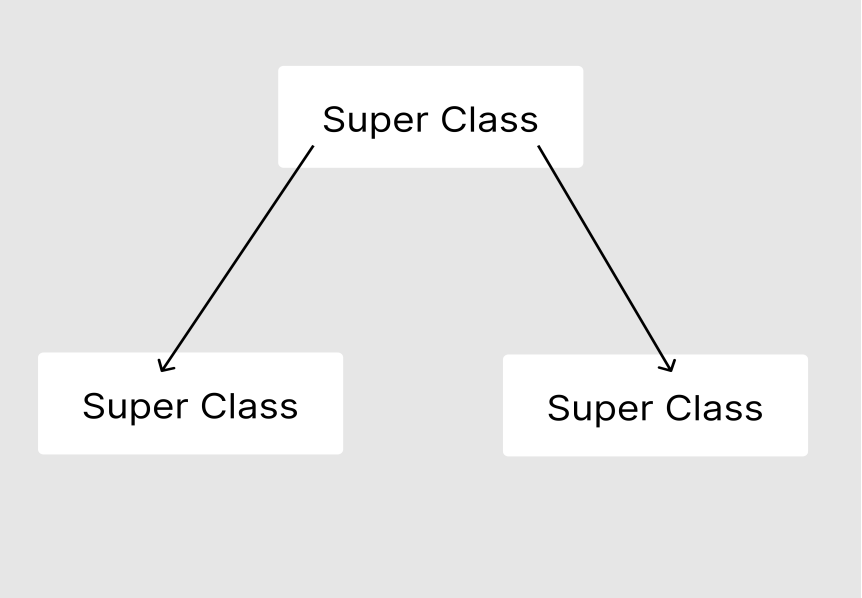
2. Multi-Level Inheritance:

- One superclass and it's one subclass, and that subclass have another subclass and same repeat again and again...



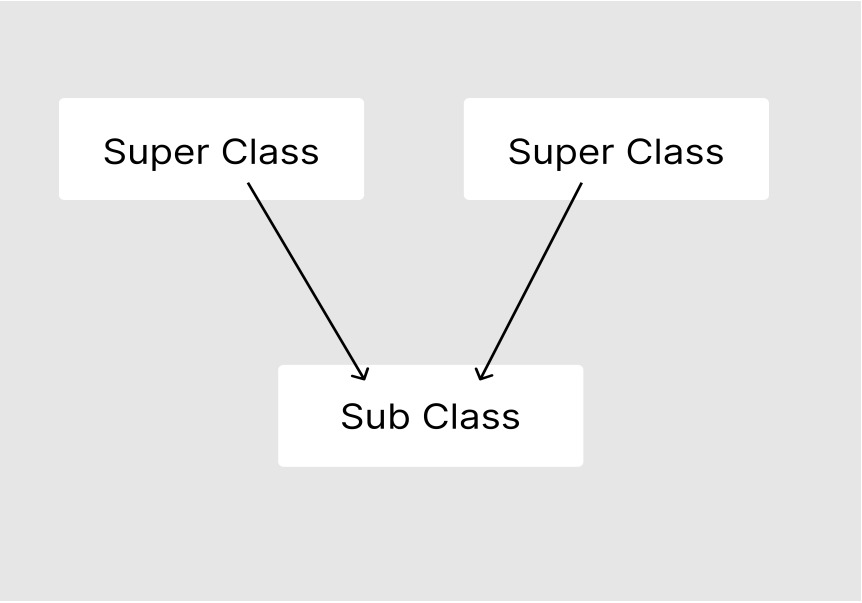
3. Heirarchical Inheritance:

- One superclass have multiple subclass.



4. Multiple Inheritance:

- It implement using interface only. Otherwise it is not possible in java.
- When we are extending the child with it's parents then it is not possible to give **comma** "," separeated names of multiple parents in case of multiple inheritance. So therefore we need to use interface to implement it.
- In this scenerio one child class aquired the properties from two parents.



Example 1

File 1

```
public class Person{
    private int age;
    private String name;

    public void setAge(int a){
        age=a;
    }

    public void setName(String n){
        name=n;
    }

    public int getAge(){
        return age;
    }

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

File 2

```
class Student extends Person{
    private int rollNo;

    public void setRollNo(int r){
        rollNo=r;
    }

    public int getRollNo(){
        return rollNo;
    }
}
```

File 3

```
public class Example{
    public static void main (String args[]){
        Student s1 = new Student();
        s1.setRollNo(100);
        s1.setName("Rajat");
        s1.setAge(24);
        System.out.println("Roll No: "+s1.getRollNo());
        System.out.println("Name: "+s1.getName());
        System.out.println("Age: "+s1.getAge());
    }
}
```

OUTPUT:
Roll No: 100
Name: Rajat
Age: 24

Example 2

File 1

```
public class Person{
    private int age;
    private String name;

    public void setAge(int a){
        age=a;
    }

    public void setName(String n){
        name=n;
    }

    public int getAge(){
        return age;
    }

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

File 2

```
class Student extends Person{
    private int rollNo;

    public void setRollNo(int r){
        rollNo=r;
    }

    public int getRollNo(){
        return rollNo;
    }
}
```

File 3

```
public class Example{
    public static void main (String args[]){
        Person s1 = new Student();
        s1.setRollNo(100);
        s1.setName("Rajat");
        s1.setAge(24);
        System.out.println("Roll No: "+s1.getRollNo());
        System.out.println("Name: "+s1.getName());
        System.out.println("Age: "+s1.getAge());
    }
}
```

OUTPUT:
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
The method setRollNo(int) is undefined for the type Person
The method getRollNo() is undefined for the type Person

- This error is occurs due to the reference of the class is Person at the time of object creation. So therefore we can't access setRollNo() and getRollNo() method.
- So the point is that if we are creating the object of subclass but the reference is of parent class then we can not access methods of child class.

Points to Remember

- In the java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclass.

- Private members of the superclass are not accessible by the subclass and can only be indirectly accessed.
- Members that have default accessibility in the superclass are also not accessible by subclass in the other package.

Chapter 25 : Java This Keyword

- The this object reference is a local variable in instance member methods refering the caller object.
- this keyowrd is used as a reference to the current object which is an instance of the current class.
- The this reference to the current object is useful in situations where a local variable hides or shadows, a field with the same name.

Example

```
class Box{
    private int l,b,h;

    public void setDimension(int l, int b, int h){
        this.l=l;
        this.b=b;
        this.h=h;
    }
}
public class Example{
    public static void mai (String args[]){
        Box box = new Box();
        box.setDimension(12,10,5);
    }
}
```

- if a method needs to pass the current object to another method, it can do so using the **this** reference.
- Note that the this reference can not be used in a static context, as static code is not executed in the context of any object.

Chapter 26 : Java Super Keyword

- In inheritance, subclass object when call an instance member function of subclass only, function contains implicit reference varibles **this** and **super** both refering to

the current objects (Object of Subclass).

The only difference in this and super is:

1. this reference variables is of subclass type.
2. super reference variable is of superclass type.

Uses of super keyword

- If your method overrides one of it's superclass methods, you can invoke the overridden method through the use of the keyword super.
- It avoids name conflit between member variables of superclass and subclass.

Example for Second Point

```
class A{
    //superclass z
    int z;

    public void f1(){}
}
class B extends A{
    // subclass z
    int z;

    public void f1(){}

    public void f2(){
        //local variable z
        int z;
        z=2;
        this.z=3;
        super.z=4;
        System.out.println("Super Class Z = "+super.z);
        System.out.println("Sub Class Z = "+this.z);
        System.out.println("Local Variable Z = "+z);
    }
}
public class Example{
    public static void main (String args[]){
        B obj = new B();
        obj.f2();
    }
}
```

OUTPUT:
Super Class Z = 4
Sub Class Z = 3
Local Variable Z = 2

Chapter 27 : Java Polymorphism

- Polymorphism is the idea that entities in code can have more than one form, is a popular concept in object-oriented programming.

There are two types to implement polymorphism in Java

1. Overloading

- If two methods of a class (whether both declared in the same class or both inherited by a class or one declared and one inherited) have **same name but different signitures(parameters)**, then the method is said to be **overloaded**.
- It is said to be **static**.
- It occurs at **compile time**.

Example

```
class A{
    public void f1(int x){
        System.out.println("Class A");
    }
}
class B extends A{
    public void f1(int x, int y){
        System.out.println("Class B");
    }
}
public class Example{
    public static void main (String args[]){
        B obj = new B();
        obj.f1(5);
        obj.f1(5,6);
    }
}
```

OUTPUT:
Class A
Class B

2. Overriding

- Method overriding is defining a method in sub-class with the **same signiture** with specific implementation in respect to subclass.
- It is said to be **dynamic**.
- It occurs at **runtime**.

Example

```
class A{
    public void f1(int x){
        System.out.println("Class A");
    }
}
class B extends A{
    public void f1(int x){
        System.out.println("Class B");
    }
}
public class Example{
    public static void main(String args[]){
        B obj = new B();
        obj.f1(5);
    }
}
```

OUTPUT:
Class B

Points to Remember

- In overriding we have to make sure the name and signitures of methods are same. And one method is in super class and another in subclass.
- In overloading we can place methods in different class or within same class.

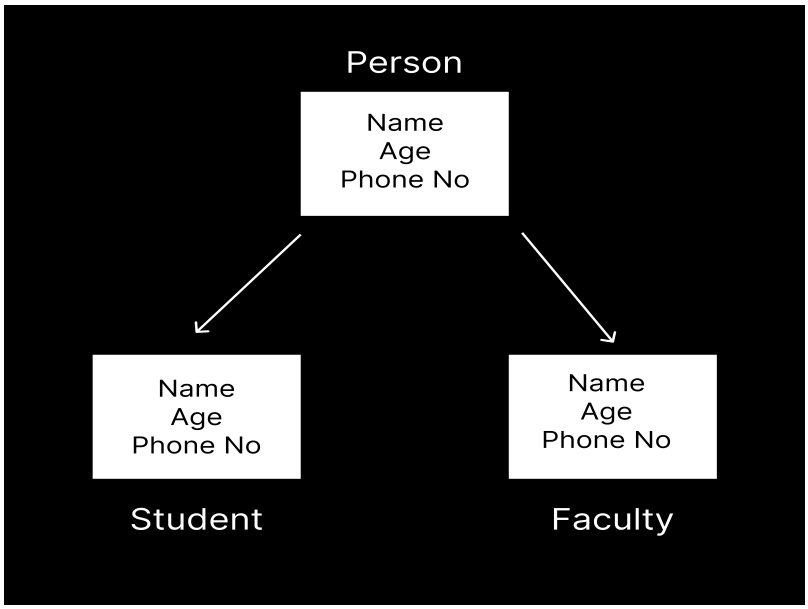
Chapter 28 : Java Abstract Keyword

- **Abstraction -** It is a process of hiding the implementation and show only functionality to the users.

Abstract Clases

- Abstract classes are declared with the abstract keyword.
- An abstract class cannot be instantiated(means we can not create object of abstract class).

Why we use abstract class



There are so many common things are between different object so we can use abstract class means that no implementation is there so whether it is Student or Faculty, it works for both by extending it.

- Java abstract classes are used to declare common characteristics of subclass.
- It can only be used as superclass for other classes that extend the abstract class.
- Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform.
- You can not create a object of abstract class but you can create a reference variable of abstract class.

Abstract Methods

- An **abstract class** can include methods that contains **no implementation**. These are called abstract methods.

- The abstract method declaration must have end with **semicolon** rather than a **block**.
- If a class has any abstract methods, whether declared or inherited the entire class must be declared as abstract.
- If a class is abstract then we can also use **non-abstract** methods.
- Static methods can't be abstract.
- Here we achieve only 50% of abstraction therefore, we introduced interface to achieve 100% abstraction.

How to use abstract class?

- We need to extends a abstract class and override methods.

Example

```
abstract class Person{
    abstract void show();
}

class Student extends Person{
    //overriding the show method in abstract class
    void show(){
        System.out.println("Working...");
    }
}

public class AbstractExample{
    public static void main(String args[]){
        Student s = new Student();
        s.show();
    }
}
```

OUTPUT:
Working...

Chapter 29 : Java Interface Keyword

- Interface begins with a keyword **interface**.
- Using interface we can achieve 100% abstraction.

- Interface just specify the method declaration **(implicitly public abstract)** and can only contains fields **(implicitly public static final)**.
- Using interface we can also achieve multiple inheritance which is not possible in java inheritance.
- We can not create an object of interface also as like abstract.
- Interface do not have constructor (we can not use super keyword).

Example

```
interface SomeName{
    int x;
    void someFunction();
}
```

To implement functions which we have declared in interface we need to use **implements** keyword.

Example

```
interface I1{
    void someFunction();
}
class A implements I1{
    public void someFunction(){
        System.out.println("Working...");
    }
}
public class InterfaceExample{
    A myObj = new A();
    myObj.someFunction();
}
```

OUTPUT:
Working...

Example of Multiple Inheritance Using Interface

```
interface A{
    public void show();
}
interface B{
    public void print();
}
class C implements A,B{
    public void show(){
        System.out.println("Display...");
    }
    public void print(){
        SYstem.out.println("Print...");
    }
}
public class MultipleIheritanceExample{
    public static void main (String args[]){
        C myObj = new myObj();
        myObj.show();
        myObj.print();
    }
}
```

OUTPUT:
Display...
Print...

Difference Between Abstract Class and Interface

- Abstract class can have any access modifiers for members.
- Interface have only public members.

Note: Abstract class is a class but Interface is not a class.

- Abstract class may or may not contain abstract methods.
- Interface can not have defined methods.

Note: Functions in abstract class are by default abstract. we don't need to use abstract keyword.

- **Abstract class** have **static** and **non static** members variables.
- Interface can have only static member variables.

Note: In interface, all the variables which we are using are by default static and functions are non static. So there is no need to use static keyword.

- **Abstract class** can have **final** or **non final variables**. **Interface** can have **only final variables**.
- Interface do not have constructor unlike abstract class.(Compiler also can't make constructor be default).

Chapter 30 : Java Constructor

- Constructor is a member function of a class.
- The name of constructor is same as the name of the class.
- Constructor has no return type.

Example

```
1  public class Box{
2      private int l,b,h;
3
4      public Box(){
5          l=10;b=20;h=30;
6      }
7
8      public static void main(String args[]){
9          Box b1 = new Box();
10     }
11 }
```

Note: Constructor is a special member function which has no need to call, just we need to create a object then it will called automatically.

Why the need of constructor?

Because, only memory allocation to the objects doesn't make them object but they are properly initialized then they are called to be objects. So this is the work of constructor.

Example

In below example we have created a instance of Example, but as you can see there is only memory allocation is done but is obejct is properly initialized?

```
public class Example{
    public static void main(String args[]){
        Example e1 = new Example();
    }
}
```

There are two types of constructors:

1. Parameterized
2. Default

Points to Remember

- A constructor is a special method that is used to initialize a newly created object and is called implicitly just after the memory is allocated for the object.
- It is not mandatory for the coder to write a constructor for the class.
- When there is no constructor defined in the class by programmer, compiler implicityly provide a default constructor for the class.
- Constructor can be parameterized.
- Constructor can be overloaded.

Constructor Inheritance in Java

- Constrctors are not inherited by subclass.

What happens when object of subclass created?

If we created a object of subclass then default constructor of subclass in called but before calling the constructor of subclass it will call superclass constructor. therefore the execution order is firstly superclass then subclass.

- Subclass's constructor invokes constructor of superclass.
- Explicit call to the superclass constructor from subclass's constructor can be made using **super()**.
- You can write subclass constructor that invokes the constructor of the superclass, either implicitly or by using the super keyword.

```
class A{
    int a;
    public A(){
    System.out.println("A");
    }
}
class B extends A{
    int b;
    public B(){
        System.out.println("B");
    }
}
public class Example{
    public static void main(String args[]){
        B obj = new B();
    }
}
```

OUTPUT:
A
B

Note: If we created a parameterized constructor in superclass then the constructor of subclass in unable to call the constructor of superclass. then we have to use super keyword in the constructor of subclass to call superclass constructor with the parameter. e.g. super(4); (we have to write super(4); in the first line of the subclass constructor otherwise it will show the error).

Constructor Chaining in Java

- Constructor can call other constructor of the same class or superclass.
- Constructor call from a constructor must be the first step(Call should be appeared in the first line).
- Such series of invocation of constructors is called as **constructor chaining**.

Example

```
class A{
    public A(){
        System.out.println("A 1");
    }
}
class B extends A{
    public B(){
        this(4);
        System.out.println("B 1");
    }

    public b(int k){
        System.out.println("B 2");
    }
}
public class Example{
    public static void main(String args[]){
        B obj = new B();
    }
}
```

OUTPUT:
A 1
B 2
B 1

Explaination:
As we know we didn't use super keyword because compiler automatically add super keyword in default constructor of subclass. But in this above example we used this keyword in constructor of subclass which calls the constructor of subclass. But we had passed one argument in this kewyword, so it will called the paramterized constructor first i.e. B 2. then in B 2 function compiler will add super keyword, therefore, it will call A 1. and the output like A 1, B 2 followd by B 1.

- First line of constructor is either super() or this() (by default super() hota hai).
- Constructor never contains super() and this() both at a time.

Chapter 31 : Java Static Keyword

Static members in Java:
1. Static Variables:

- static variables are declared in the class using static keyword.
- static variables are by default initialized to its default value.
- static variables has a single copy for the whole class and doesw not depend on the objects.

Note:

1. Static member functions have only access to the static variables.
2. To call static members use class name and later use dot(.) i.e. Example.Test

Example

```
public class Example{
    //instance variable
    int x;

    // static variable
    static int y;

    //instance function
    public void fun1(){}

    //static function
    public static void fun2(){
        y=4;
    }

    //static inner class
    static class Test{
        public static String Country = "INDIA";
    }
    public static void main(String args[]){
        Example ex1 = new Example();
        Example ex2 = new Example();
        Example.fun2();
        System.out.println(Example.Test.Country);
    }
}
```

OUTPUT:
INDIA

2. Static Function:
(See above example for reference)

3. Static Class:

- We can have a class inside a class which is known as inner class.
- inner class can be qualified with the keyword static.
- We can not declare static variables inside the method or function.
- But we have static inner class.

Static members Inheritance In Java

- A class C inherits from its direct superclass all concreate methods M(Both static and instance) of the superclass.
- No method declared in C has same signiture.

Example

```
class Parent{
    public static void f1(){
        System.out.println("Hello");
    }
}
class Child extends Parent{

}
public class Example{
    public static void main(String args[]){
        Child.f1();
    }
}
```

OUTPUT:
Hello

```
class Parent{
    public static void f1(){
        System.out.println("Hello");
    }
}
class Child extends Parent{
    public static void f1(){
        System.out.println("Yo! Man");
    }
}
public class Example{
    public static void main(String args[]){
        Child.f1();
    }
}
```

OUTPUT:
Yo! Man

If subclass has a method M with the same signiture as of the method present in the super class, then method M hides the method of superclass. (Called it as function hiding when we use static keyword, but if we remove the static keyword then is behave like overriding).

In above example child's function hides the function of it's parent due to static keyword.

Note: In such case, if we declared a static method in parent class then we have to use static method in child class also. We can not do one static and another one is non static.

Points to Remember

- It is a compile-time error if a static method hides an instance method.
- It is a compile-time error if an instance method override a static method.
- Static member variables do not inherit but functions inherit but functions can inherits but with same parameters.

Chapter 32 : Java Import Keyword

- import is a keyword in java.
- It is used to import classes of other packages.

Example

File 1

```
package pack2;
public class Student{
    private int rollNo;
    private String name;

    public void setRollNo(int r){
        rollNo=r;
    }

    public void setName(String n){
        name=n;
    }

    public int getRollno(){
        return rollNo;
    }

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

File 2

```
package pack1;
import pack2.Student;

public class Example{
    public static void main(String args[]){
        Student s1 = new Student();
        s1.setRollNo(100);
        s1.setName("Rajat");
        System.out.println("Roll No : "+s1.getRollNo());
        System.out.println("Name : "+s1.getName());
    }
}
```

OUTPUT: Command-Line Execution
javac -d . Student.java
javac -d . Example.java
java pack1.Example
Roll No : 100
Name : Rajat

Chapter 33 : Java Initialization Block

There are two types of initialization block:
1. Instance initialization block:

```
public class Example{
    private int x;

    {
        System.out.println("Initialization Block: x ="+x);
        x=5;
    }

    public void Example(){
        System.out.println("Constructo: X ="+x);
    }

    public static void main(String args[]){
        Example e1 = new Example();
        Example e2 = new Example();
    }
}
```

OUTPUT:
Initialization Block: X = 0
Initialization Block: X = 0

Note: Instance initialization block runs automatically when we created a object but it have more preference than the constructor. thats why we get the value 0 both the times.

- An intance initializer or initialization block declared in a class is executed when an instance of the class is created.
- return keyword can not be used in initialization block.

- Instance initializers are permitted to refer to the current object via the keyword **this** and to use the keyword **super**.

2. Static initialization block:

```
public class Example{
    private static int k;

    static{
        System.out.println("Static Initialization Block: K ="+k);
        k=10;
    }

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

OUTPUT:
Static Initialization Block: K = 0

- A static initializer declared in a class is executed when the class is initialized.
- Static initializers may be used to initialize the class variables of the class.
- this or super can not be used in static block.
- return keyword can not used in static initialization block.

Note: Static block runs before the object creation. And static block runs once for all the objects that we are creating but instance inistialization block ruhns freshly for every new object.

Chapter 34 : Java Final Keyword

The final keyword used in :
1. final instance variable:

```
public class Example{
    //final instance variable
    private final int x;

    Example(){
        x=5;
    }
    public static void main (String args[]){
        Example e1 = new Example();
    }
}
```

- A java variable can be declared using the keyword final, then the final variable can be assigned only once.
- A varibale that is declared as final and not initialized is called a blank final variable.
- A blank final varibale forces either the constructor to initialize it or initialization block to do this job.

2. final static variable:

```
public class Example{
    //final static varibale
    private final static int x;

    static{
        x=5;
    }
    public static void main(String args[]){
        Example e1 = new Example();
    }
}
```

- Static member variable when qualified with a final keyword, it becomes blank until initialized.
- Final static variable can be initialized during declartion or within the static block.

3. final local variable:

```
public class example{
    public void fun(){
        final int k; // final local variable
    }
    public static void main (String agrs[]){
        Example e1 = new Example();
    }
}
```

- Local variables that are final must be initialized before it's use, but you should remember this rule is applicable to non final local variables too.
- once they are initialized, can not altered.

4. final class:

```
//final class
final class Dummy{}
public class Example{
    public static void main(String args[]){

    }
}
```

- Once we created a final class so we cannot create subclass of it. means that there is inheritance is not possible **(No inheritance)**.

5. final methods:

```
class Dummy{
    //final method
    public final void someFunction(){

    }
}
class MoreDummy extends Dummy{
    // show error
    public void someFunction(){

    }
}
public class Example{
    public static void main(String agrs[]){

    }
}
```

- methods declared as final can not be overridden.

Chapter 35 : Java Garbage Collector

- In other languages like C++, programmer is responsible to create a new object and to destroy the object. Usually programmertaking very much care to create new objects and neglecting distribution of useless object, because of his neglegance at a certain point for creation of new objects, sufficient memory not available (because total memory fill with useless objects only) and total application will be down. Hence Out-Off-Memory-Error is very common problem in old languages like C++.
- But in Java Programming, Programmer only responsible for creation of objects and not to destroy the useless objects. Sun's (Sun Microsystems) people provide one assistant to destroy useless objects.
- This assitance is always runningin background(Demon Thread) and destroy useless objects. Due to this chance of failing program with memory problem is very very low. This assistance is called **Garbage Collector**.

The ways to make an object eligible for GC:

- Even though programmer is not responsible to destroy useless object, it is highly recommended to make an object eligible for GC, if it is no longer required.
- An object is said to be eligible for GC, if and only if it doesn't contain any reference variable.

The following are various ways to make object eligible for GC:

1. Nullifying the reference variables:

if an object is no longer required then assign them null to all it's references then it automatically eligible for GC.

```
Student s1 = new Student();
Student s2 = new Student();

// if they are not reuired anymore just null them;
s1=null;
s2=null;
```

2. Re-Assigning the Reference Variables:

If an object is no longer required then Re-Assign it's reference variable to any other object then old object is automatically eligible for GC.

3. Object Created Inside a Method:

Objects created inside a methods are by default eligible for garbage collection, Once method is completed.

The methods for Requesting JVm to Run Garbage Collector:

1. Using System Class

```
System.gcc(); // static method of system class
```

2. Using Runtime Class:

```
Runtime r = Runtime.getRuntime();
r.freeMemory(); // free memory in the heap
r.totalMemory(); //total memory of the heap
r.gc(); //Requesting JVM to run GC
```

4. Island of Isolation:

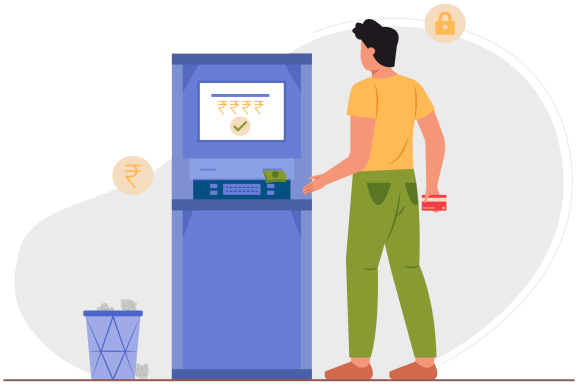
The "Island of Isolation" happens when a group of objects only reference each other, but nothing else in the program refers to them. This means they're isolated from the rest of the program, but they still take up memory because they're not being removed by the garbage collector. It's like a little island of useless objects floating around in your program's memory, taking up space.

Chapter 36 : Java Exception Handeling

Introduction to Exeception Handeling

Exceptions in java are any abnormal, unexpected events or extraordinary conditions that may occurs at runtime.

Example



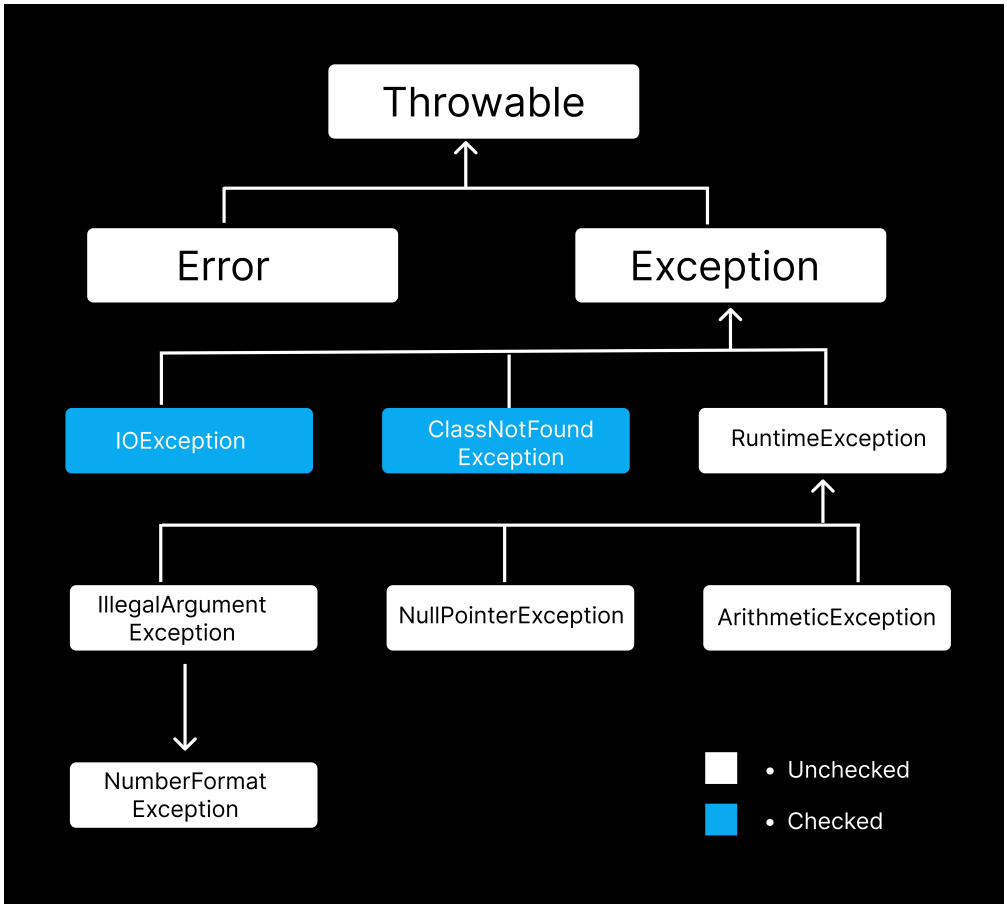
Consider the example of ATM Machine, we can set some exceptions like if there is no money in the ATM Machine or if user reached to it's daily limit then, if user want to withdraw some money from his/her account so then we have to show some arror messages to them.

There are four options to handle exceptions:

1. Default throw and Default catch
2. Default throw and Our catch
3. Our throw and Default catch
4. Our throw and Our catch

Exception Handeling

Java exception handeling is used to handle error conditions in a program systematically by taking the necessary action.



try and catch

Exceptions are handled using try and catch keywords.

Throwable

- The Throwable class provides a string variable that can be set by the subclass to provide a detailed message that provides more information of the exception occurred.
- All classes of Throwables define a one parameter constructor that takes a string as the detailed message.

Methods:

1. addSuppressed(throwable exception) ---> R.T. void
2. fillInstackTrace() ---------> R.T. Throwable
3. getCause() ----> R.T. Throwable
4. getlocalizedMessage() -------> R.T. String
5. toString() ------> R.T. String
6. getMessage() ----------> R.T. String
7. initCause()

The class Exception represents exceptions that a program faces due to abnormal or special conditions during execution.

The Exception are of two types:

1. Unchecked Exceptions
2. Checked Exceptions

Unchecked Exception Handeling

- **Unchecked Exceptions** are **Run Time Exceptions** and any of its subclasses.
- ArrayIndexOutOfBounds, NullPointerException and so on are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class.

Default throw and Our Catch

Syntax

try{
//exception which we have to handle
}
catch(<Exception type> <parameter>){
//statement
}
finally{
//finally block statement
}

Example

```
1  public class Example{
2      public static void main(String args[]){
3          try{
4              System.out.println("Result: "+3/0);
5              System.out.println("In Try");
6          }
7          catch(ArithmeticException e){
8              System.out.println("Exception: "+e.getMessage());
9          }
10         System.out.println("Hello");
11     }
12 }
```

OUTPUT:
Exception: / by zero
Hello

Note:

- For each try block there can be zero or more catch blocks, but only one finally.
- The catch blocks and finally block must along appear in conjugation with a try block.
- A try block must be followed by either least one catch or one finally block.
- The order exceptions handlers in the catch block must be from the most specific exceptions.
- In try block, if there is any exception found after that line no code will be executed after that line. i.e. System.out.println("In Try");
- If exception is type of Arithmetic then we have to use ArithmeticException only. But if we use any other class then JVM runs the default catch mechanism.
- If we create our catch block then code will be not stop but if default catch mechanism is executed then code will be stop.
- If we created multiple catch blocks then it will verified every catch method which is suitable for the try block.

Explicit Throw An Exception

- A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.

Syntax:
throw<ThrowableInstance>

- The execution reference must be of type Throwable class or one of its subclass.
- A detailed message can be passed to the constructor when the exception object is created.

Our throw and Default catch

```
public class Example{
    public static void main(String args[]){
        int balance = 5000;
        int withdrawlAmount = 6000;

        if(balance<withdrawlAmount)
            throw new ArithmeticException("Insufficient Balance");

        balance = balance-withdrawlAmount;
        System.out.println("Transaction Successfully Completed");
        System.out.println("Program Continue...");

    }
}
```

OUTPUT:
Exception in thread "main" java.lang.ArithmeticException: Insufficient Balance
atExample.main(Example.java.7)

Our throw and Our Catch

```
public class Example{
    public static void main(String args[]){
        int balance = 5000;
        int withdrawlAmount = 6000;
        try{
            if(balance<withdrawlAmount){
                throw new ArithmeticException("Insufficient Balance");
            }
            balance = balance-withdrawlAmount;
            System.out.println("Transaction Successfully Completed");
        }
        catch(ArithmeticException e){
            system.out.println("Exception: "+e.getMessage());
        }
        System.out.println("Program Continue...");
    }
}
```

OUTPUT:
Exception: Insufficient Balance
Program Continue...

Why should we throw an exception object?

- Because we want to set a different message.
- Because java cannot recognise exceptional situations of business logic.

Use of Throws in Checked Exception

- Compile time Errors in checked exceptions
- Checked exceptions forces programmers to deal with the exception that may thrown.
- IOException, SQLException, IlleagalthreadStateException,etc are checked exceptions.
- "Checked" means they will be checked at compile time itself.

Note:
To handle checked exceptions we have to use **throws** keyword or we have to handle them with try catch blocks.

Example

```
1  import java.io.*;
2
3  public class Example{
4      public static void main(String args[]) throws IOException{
5          throw new IOException();
6          System.out.println("After Exception");
7      }
8  }
```

OUTPUT:
Exception in thread "main" java.lang.Error: Unresolved compilation problem: uneacheble code at example.main(Example.java:5)

But if we write ArithmeticException instead of IOException then also it will not show error because ArithmeticException is unchecked exception.

Throws

A thows clause can be used in the method prototype.

```
method() throws <ExceptionType1>,...<ExceptionTypeN>{

}
```

The throws keyword in java programming language is applicable to a method to indicate that the method raises perticular type of exception while being processed.

The trhwos keyword in java programming langauge takes arguments as a list of the object of type java,lang.Throwable class.

Custom Exception Class Creation:

Example

```
1  package com.rajatkotangale;
2
3  public class CustomizedException {
4          public static void main(String[] args) {
5                  int age = 17;
6                  try {
7                          if(age<=18){
8                                  throw new AgeException("Underage....");
9                          }
10                 }
11                 catch(AgeException e) {
12                         System.out.println(e.toString());
13                 }
14                 catch(Exception e){
15                         System.out.println(e.toString());
16                 }
17         }
18 }
19
20 class AgeException extends Exception{
21         public AgeException(String str) {
22                 super(str);
23         }
24 }
```

OUTPUT:
com.rajatkotangale.AgeException: Underage...

Chapter 37 : Java Scanner Class

- We can read java input from System.in (http://System.in) using Scannner Class.
- Scanner is final class, that cannot be extended.
- Scanner class is a part of java.util package.

Methods:

1. next()
2. nextLine()
3. nextBoolean()
4. nextByte()
5. nextDouble()
6. nextFloat()
7. nextInt()
8. nextLong()
9. nextShort()

- A scanner breaks its input into tokens using a delimiter (ending mark of data) pattern, which by default matches whitespaces.
- The resulting tokens may then be converted into values of different types using the various next methods.

Example

```
import java.util.Scanner;

public class Example{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Your Name :");
        String name = sc.nextLine();
        System.out.println("Enter Your Age :");
        int age = sc.nextInt();
        System.out.println("Name: "+name);
        System.out.println("Age: "+age);
    }
}
```

OUTPUT:
Enter Your Name :
Rajat
Enter Your Age :
24
Name: Rajat
Age: 24

Chapter 38 : Java Enum

- An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).
- To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma.
- Note that they should be in uppercase letters.

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}

public class Main {
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
}
```

OUTPUT:
MEDIUM

```
public class Main {
  enum Level {
    LOW,
    MEDIUM,
    HIGH
  }

  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
}
```

OUTPUT:
MEDIUM

Difference between Enums and Classes

- An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).
- An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Chapter 39 : Java Regex

- A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.
- A regular expression can be a single character, or a more complicated pattern.
- Regular expressions can be used to perform all types of text search and text replace operations.
- Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions.

The package includes the following classes:

- Pattern Class - Defines a pattern (to be used in a search)
- Matcher Class - Used to search for the pattern
- PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

Example

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE)
    Matcher matcher = pattern.matcher("Visit W3Schools!");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
}
```

OUTPUT:
Match Found

Flags

Flags in the compile() method change how the search is performed. Here are a few of them:

- Pattern.CASE_INSENSITIVE - The case of letters will be ignored when performing a search.
- Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- Pattern.UNICODE_CASE - Use it together with the CASE_INSENSITIVE flag to also ignore the case of letters outside of the English alphabet

Regular Expression Patterns

The first parameter of the Pattern.compile() method is the pattern. It describes what is being searched for.

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find one character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
|	Find a match for any one of the patterns separated by
.	Find just one instance of any character
^	Finds a match as the beginning of a string as in: ^Hello
$	Finds a match at the end of the string as in: World$
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n
n{x}	Matches any string that contains a sequence of X n's
n{x,y}	Matches any string that contains a sequence of X to Y n's
n{x,}	Matches any string that contains a sequence of at least X n's

Chapter 40 : Java Lambda

- A lambda expression is a short block of code which takes in parameters and returns a value.
- Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Syntax:

Lambda expression contains a single parameter and an expression:

```
parameter -> expression
```

More than one parameter, wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

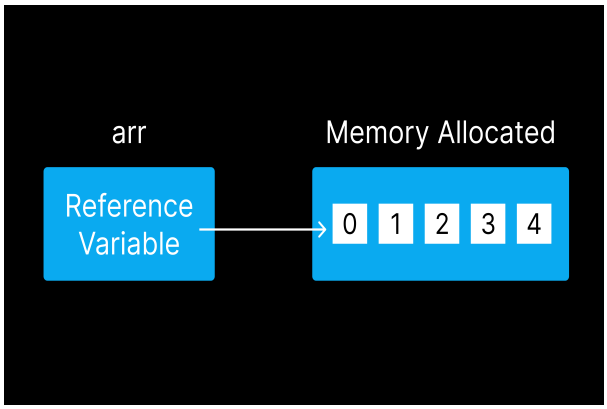
```
1  import java.util.ArrayList;
2  import java.util.function.Consumer;
3
4  public class Main {
5    public static void main(String[] args) {
6      ArrayList<Integer> numbers = new ArrayList<Integer>();
7      numbers.add(5);
8      numbers.add(9);
9      numbers.add(8);
10     numbers.add(1);
11     Consumer<Integer> method = (n) -> { System.out.println(n); };
12     numbers.forEach( method );
13   }
14 }
```

OUTPUT:
5
9
8
1

Chapter 41 : Java Arrays

There are two ways to declare arrays in Java:

1. int []arr = new int[5];
2. int arr[] = new int[5];



int arr[] = new int[]{2,4,6,8,10}; //correct

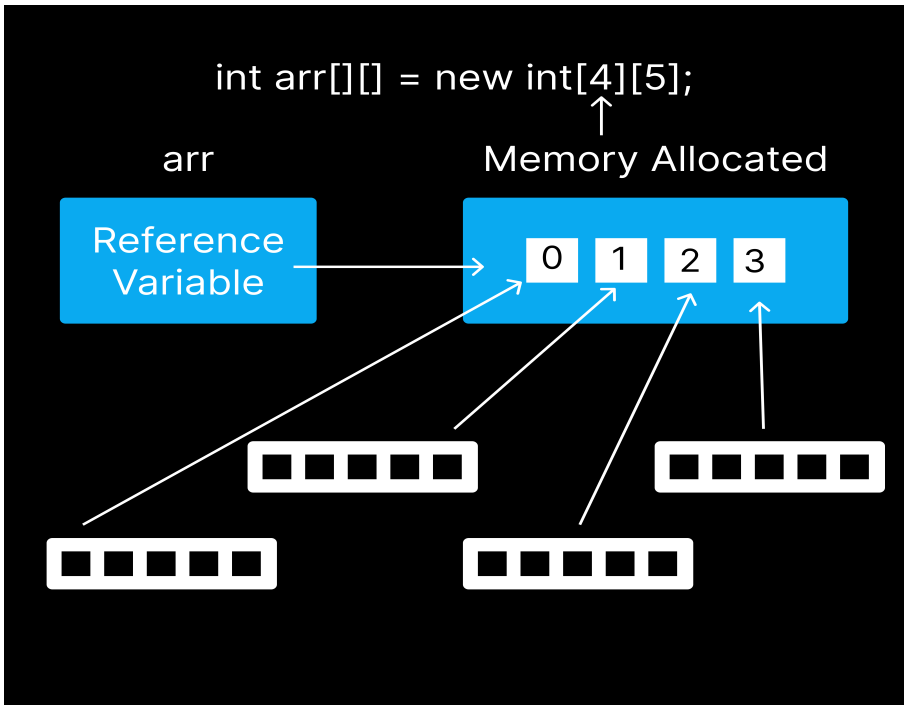
int arr[] = new int[2]{2,4,6,8,10}; //show error
We can not mentioned array size and values simultaneously.

int arr[] = {2,4,6,8,10}; //correct

Two Dimensional Array

int arr[][] = new int[4][5]; //correct
But,
int [][]arr = new int[][]; //error
int [][]arr = new int[][5]; //error
int [][]arr = new int[4][]; //correct

Example Exaplaination



Array Class

1. Sorting Methods:

- **sort(T[] a):** Sorts the specified array of objects into ascending order.
- **sort(T[] a, Comparator<? super T> c):** Sorts the specified array of objects according to the order induced by the specified comparator.
- **sort(int[] a):** Sorts the specified array of ints into ascending numerical order.
- **sort(int[] a, int fromIndex, int toIndex):** Sorts the specified range of the array of ints into ascending numerical order.

Similar methods exist for other primitive types (byte, short, long, float, double, char) and for arrays of those types.

2. Searching Methods:

- **binarySearch(T[] a, T key):** Searches the specified array of objects for the specified object using the binary search algorithm.
- **binarySearch(T[] a, int fromIndex, int toIndex, T key):** Searches a range of the specified array of objects for the specified object using the binary search algorithm.
- **binarySearch(int[] a, int key):** Searches the specified array of ints for the specified value using the binary search algorithm.

3. Equality and Comparison Methods:

- **equals(T[] a, T[] a2):** Returns true if the two specified arrays of objects are equal to one another.
- **equals(int[] a, int[] a2):** Returns true if the two specified arrays of ints are equal to one another.
- **deepEquals(Object[] a1, Object[] a2):** Returns true if the two specified arrays are deeply equal to one another.

4. Fill Methods:

- **fill(T[] a, T val):** Assigns the specified value to each element of the specified array of objects.
- **fill(T[] a, int fromIndex, int toIndex, T val):** Assigns the specified value to each element of the specified range of the array of objects.
- **fill(int[] a, int val):** Assigns the specified value to each element of the specified array of ints.
- **fill(int[] a, int fromIndex, int toIndex, int val):** Assigns the specified value to each element of the specified range of the array of ints.

5. Copy Methods:

- **copyOf(T[] original, int newLength):** Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
- **copyOfRange(T[] original, int from, int to):** Copies the specified range of the specified array into a new array.

6. Other Methods:

- **toString(T[] a):** Returns a string representation of the contents of the specified array.

Chapter 42 : Java Generics

Generics

- It is an idea to allow reference types to be parameterized to methods and classes.

Types of Generics

1. Generic Methods:

- Set the placeholder < any alphabet >.

- Each type parameter section contains one or more type parameters separated by commas.
- The type parameters can be used to declare the return type.
- Type parameters can represent only reference type, not primitive types.

Normal Method -

Here we need to create new fuction for every new datatype. So to resolve this probem we have to create a generic method.

```
public class GenericsMethod {
    public void printArray(String str[]){
        for(int i=0;i<str.length;i++){
            System.out.println(str[i]);
        }
    }

    public void printArray(Integer nums[]){
        for(int i=0;i<nums.length;i++){
            System.out.println(nums[i]);
        }
    }

    public void printArray(Float decimalnums[]){

        for(int i=0;i<decimalnums.length;i++){
            System.out.println(decimalnums[i]);
        }
    }

    public static void main(String[] args) {
        GenericsMethod myObj = new GenericsMethod();

        String fruits[] = new String[]{"Apple","Banana","Chiku"};
        Integer nums[] = new Integer[]{1,2,3,4,5};
        Float decimalnums[] = new Float[]{2.33f,5.55f,6.44f};

        myObj.printArray(fruits);
        myObj.printArray(nums);
        myObj.printArray(decimalnums);
    }

}
```

OUTPUT:

Apple
Banana
Chiku
1
2
3
4
5
2.33
5.55
6.44

Generic Method -

```
public class GenericsMethod {
    //generic method to generalize all the reference data type
    public <T> void printArray(T str[]){
        for(int i=0;i<str.length;i++){
            System.out.println(str[i]);
        }
    }

    public static void main(String[] args) {
        GenericsMethod myObj = new GenericsMethod();

        String fruits[] = new String[]{"Apple","Banana","Chiku"};
        Integer nums[] = new Integer[]{1,2,3,4,5};
        Float decimalnums[] = new Float[]{2.33f,5.55f,6.44f};

        myObj.printArray(fruits);
        myObj.printArray(nums);
        myObj.printArray(decimalnums);

    }
}
```

OUTPUT:
Apple
Banana
Chiku
1
2
3
4
5
2.33
5.55
6.44

2. Generic Classes -

- Set the placeholder < any alphabet >.
- Each type parameter section contains one or more type parameters separated by commas.
- The type parameters can be used to declare the return type.
- Type parameters can represent only reference type, not primitive types.

Normal Class -

Here we need to create different classes for every data type to implement the below program. So therefore we need to generic class to reuse the same code.

```
class DataCheck{
    Integer i;

    public void setData(Integer num){
        i=num;
    }

    public Integer getData(){
        return i;
    }
}

public class GenericsClass {
    public static void main(String[] args) {
        DataCheck dc = new DataCheck();
        Integer i1 = new Integer(10);
        dc.setData(i1);
        System.out.println(dc.getData());
        setData("Rajat"); // if we need to set this data we have to create a
                            whole new class for string data type.
    }

}
```

Generic Class -

```
//generic class to generalize a whole class

class DataCheck<E>{

    E i;

    public void setData(E num){
        i=num;
    }

    public E getData(){
        return i;
    }
}

public class GenericsClass {
    public static void main(String[] args) {
        DataCheck<Integer> d1 = new DataCheck<Integer>();
        DataCheck<String> d2 = new DataCheck<String>();

        d1.setData(10);
        d2.setData("Rajat");

        System.out.println(d1.getData());
        System.out.println(d2.getData());
    }
}
```

OUTPUT:
10
Rajat

Advantages of Generics

- Code reusability
- Improved readability
- Improved performance

Chapter 43 : Java Collection Framework

Collection -

- Collection is an interface which extends the different child interfaces.

- It is extended by Iterable interface.

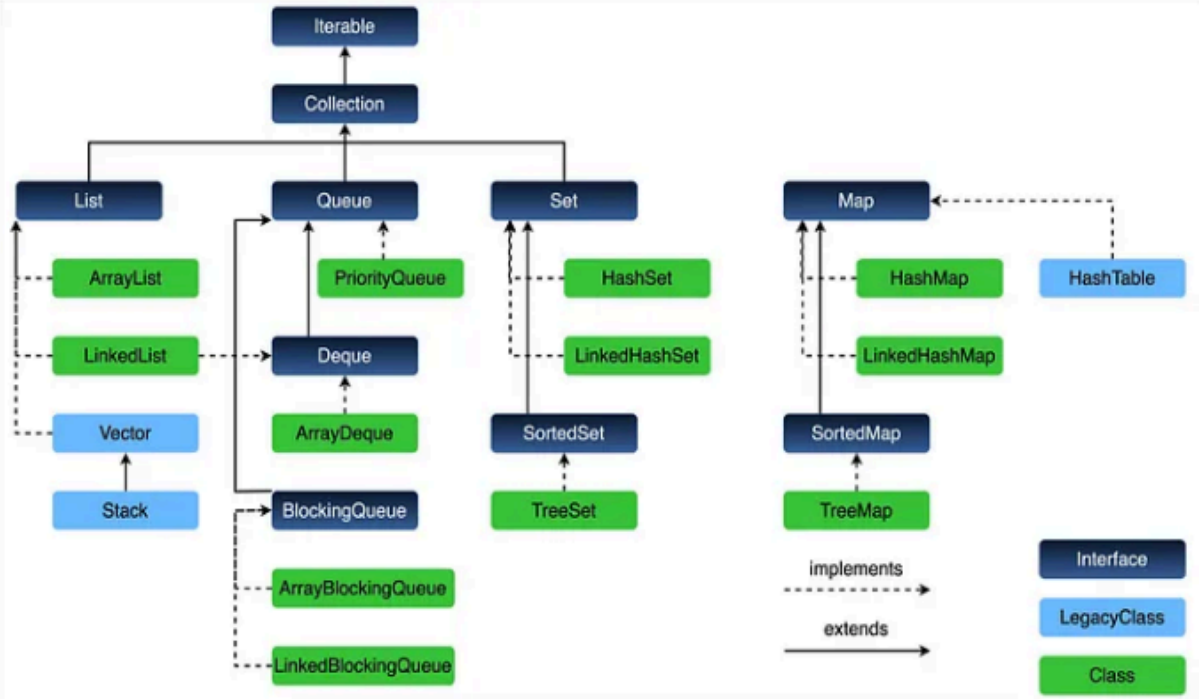
Collections -

- Collections is a utility class which includes static methods to perform different types of manipulation on the collections.

Why do we need Collection?

- To perform data manipulations like deletion, insertion, updating, sorting and searching etc.

Flow Chart Of Collection Framwork



Iterable Interface -

- It includes a single method i.e. iterator() which is used to iterate the collections.

Iterator Interface -

Methods:

- hasNext() ––––– return type boolean
- next() –––––– return type E (element)
- remove() ––––– return type void

Collection Interface -

- Extends Iterable interface i.e. public interface Collection<E> extends Iterable<E>

Methods:

- add(E e) ——— return type boolean
- addAll(Collection name) ——— return type boolean
- clear() ——— return type void
- contains(object) ——— return type boolean
- containsAll(Collection name) —— return type boolean
- equals(object) ——— return type boolean
- hashCode() —— return type int
- isEmpty() ——— return type boolean
- iterator() ———- return type Iterator<E>
- remove(object) ——— return type boolean
- removeAll(Collection name) —— return type boolean
- retainAll(Collection name) —— return type boolean
- size() ——— return type int
- toArray() ——- return type object[]

List Interface -

- Extends Collection interface.

Methods:

- All methods of collection interface and iterable interface.
- add(int index, E element) —— return type void
- addAll(int index, Collection name) —— return type boolean
- get(int index) ——— return type E
- indexOf(object o) ——- return type int
- lastIndexOf(object o) –- return type int
- listIterator() ———- return type ListIterator<E>
- listIterator(int index) ———- return type ListIterator<E>
- remove(int index) ———- return type E
- set(int index, E element) —— return type E
- sublist(int fromIndex, int toIndex) –- return type List<E>

ArrayList Class -

- Implements list interface.
- Allow heterogenous data <Object>
- Similar to array but there is no size limit
- Allow duplicates
- Allow null values
- Insertion order will be managed
- Default size is 10 with no load factor and increased by (currentSize*3/2)+1

Declaration

```
ArrayList<Integer> arr = new ArrayList<Integer>();
```

Declaring variable arr of type ArrayList<Integer> and initialize it with a new instance of ArrayList<Integer> using default constructor

Methods:

- All methods of iterable, collection, list interface
- clone() ——— return type object
- removeRange(int fromIndex, int toIndex) ——— return type void
- trimToSize() ——— return type void
- ensureCapacity(int MinCapacity) ——— return type void

LinkedList Class -

- Implements list interface and deque interface both.
- Allow heterogenous data
- Insertion order is managed but data is not stored in the sequential manner as like ArrayList.
- Allow null values
- Allow duplicates
- Default size empty i.e. it starts empty
- Internally it uses the doubly linked list data structure

Methods:

- All methods of iterable, collection, list and deque interface.

Vector Class -

- It is a legacy class i.e. introduced in 1st version of java.
- Same as ArrayList but it has some extra methods that collection does not contain.
- Slower operation than ArrayList. Due to Synchronization.
- All methods of vector class are synchronized so therefore one process running at time. So it affects the performance of the vector.
- Allow null values
- Allow duplicates
- Insertion order managed
- Default size is 10 and increases by 2X.
- Allow heterogenous data.

Methods:

- All methods of iterable, collection and list interface.
- capacity() ——— return type int
- addElement(E element) ——— return type void
- clone() ———————— return type object
- copyInto(array name) ————- return type object (array must be of object type)
- elementAt(int index) ———- return type E
- elements() ———————— return type Enumeration<E>
- ensureCapacity(int MinCapacity) ——— return type void
- firstElement() ———————— return type E
- insertElementAt(E, int index) —— return type void
- lastElement() ——— return type E
- lastIndexOf(Object)
- removeElement(Object) ————- return type boolean
- removeElementAt(int index) ———- return type void
- removeRange(int fromIndex, int toIndex) ———- return type void
- removeAllElements() ————- return type void
- setElementAt(E, int index) ——— return type void
- toString() ——— return type String
- trimToSize() ——— return type void

Stack Class -

- Extends vector class
- Allow duplicates
- Allow null values
- Heterogenous data allowed
- Default size is 10 and increases by 2x.

Methods:

- All methods of vector.
- empty() ————————— return type boolean
- peek() —————————— return type E
- pop() ——————————- return type E
- push(E) —————————— return type E
- search(Object) ——— return type int

Queue Interface -

- Extends collection interface and iterable interface

Methods:

- All methods of Collection and interable interface.
- element() ——— return type E
- offer(E) ——— return type boolean
- peek() ——— return type E
- poll() ———- return type E
- remove() ——- return type E
- add(E) ———- return type boolean
  ... and so on

Note

- offer(),peek(),poll() these functions returns false if any problem is occurred but in case of element(),add(),remove() they show exception if any problem is occurred.

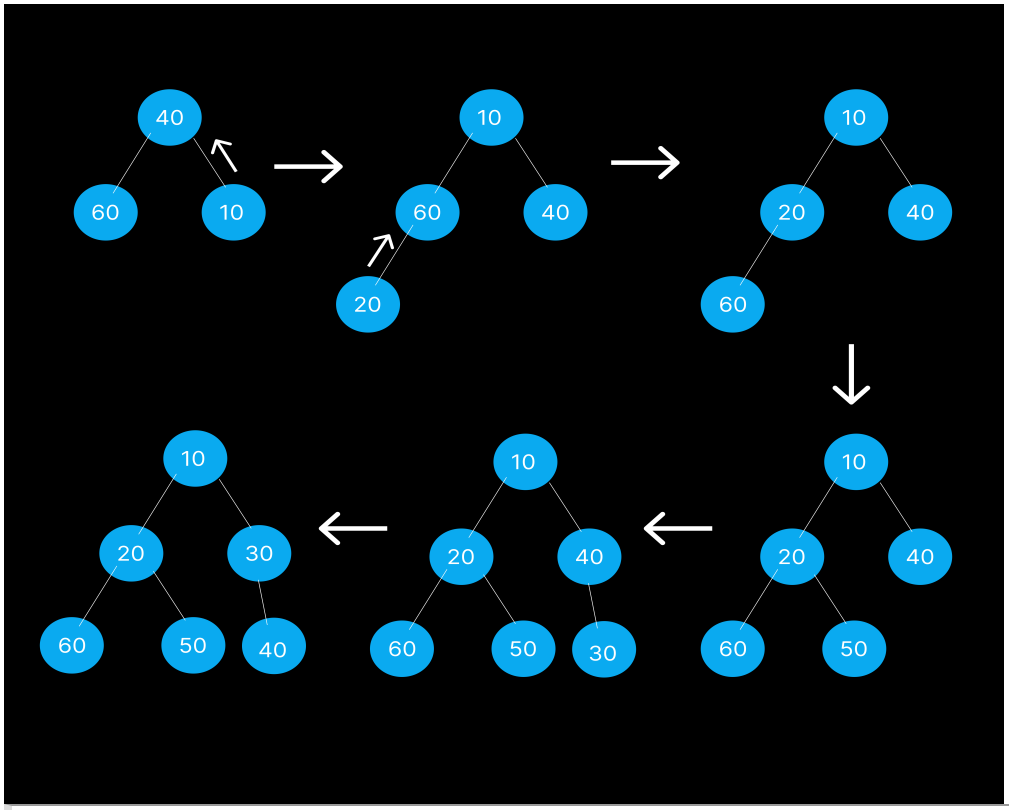
PriorityQueue Class -

- Implements queue interface

- Doesn’t allow null values.
- Implements min heap data structures means that smaller element place at 1st position.

Min Heap: Lowest element at the topmost position.

Elements : 40, 60, 10, 20, 50, 30



Final Order: 10, 20, 30, 60, 50, 40

Methods:

- All the methods of Iterable, Collection, Queue interface.
- comparator() ——— return type comparator

To Implement **Max Heap** just need to reverse the order:

PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseorder());

Deque Interface -

- Extends queue interface

Methods:

- All methods of Iterable, collection, Queue interface.

- addFirst(E) ——— return type void
- addLast(E) ——— return type void
- descendingIterator() ——— return type Iterator<E>
- element() ——— return type E
- getFirst() ——— return type E
- getLast() ——— return type E
- iterator() ——— return type E
- offer(E) ——— return type boolean
- offerFirst(E) ——— return type boolean
- offerLast(E) ——— return type boolean
- peek() ——— return type E
- peekFirst() ——— return type E
- peekLast() ——— return type E
- poll() ——— return type E
- pollFirst() ——— return type E
- pollLast() ——— return type E
- pop() ——— return type E
- push(E) ——— return type void
- removeFirst() ——— return type E
- removeLast() ——— return type E
- removeFirstOccurrence(object) ——— return type boolean
- removeLastOccurrence(object) ——— return type boolean

ArrayDeque Class -

- Double ended queue data structure is used.
- We can add and remove elements from both the ends.
- Null values are not allowed.

Methods:

- All the methods from Iterable, Collection, Queue and Deque interface.

SET Interface -

- Extends collection interface

Methods:

- All methods from Iterable and Collection interface

HashSet Class -

- Duplicates not allowed
- Null allowed
- Insertion order is not maintained.
- HashSet used Hash Table data structure
- Default size is 16
- Load factor is 0.75
- Increases by 2X

Hash Table:

Using hashing we need to place the elements.

Data: 10, 8, 25, 16, 20, 18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16		18		20				8	25	10					

We need to decide the element position on the basis of mod (remainder).

We have to divide a specific element with current capacity of the hashmap.

If we take a first element as 10. So the calculation is like 10/16, so we get remainder as 10. Therefore the index position of the 10 is 10th index.

... and so on.

> Final Order : 16, 18, 20, 8, 25, 10

Methods:

- All methods from Iterable, Collection and Set interface.

LinkedHashSet Class -

- Insertion order is maintained.
- It is a combination of LinkedList and HashSet.
- Otherwise, everything is the same as HashSet.

Methods:

- All same as HashSet

SortedSet Interface -

Methods:

- comparator() ——————— return type comparator
- first() ——————————— return type E
- last() ——————————— return type E
- subset(E fromElement, E toElement) ——————— return type SortedSet

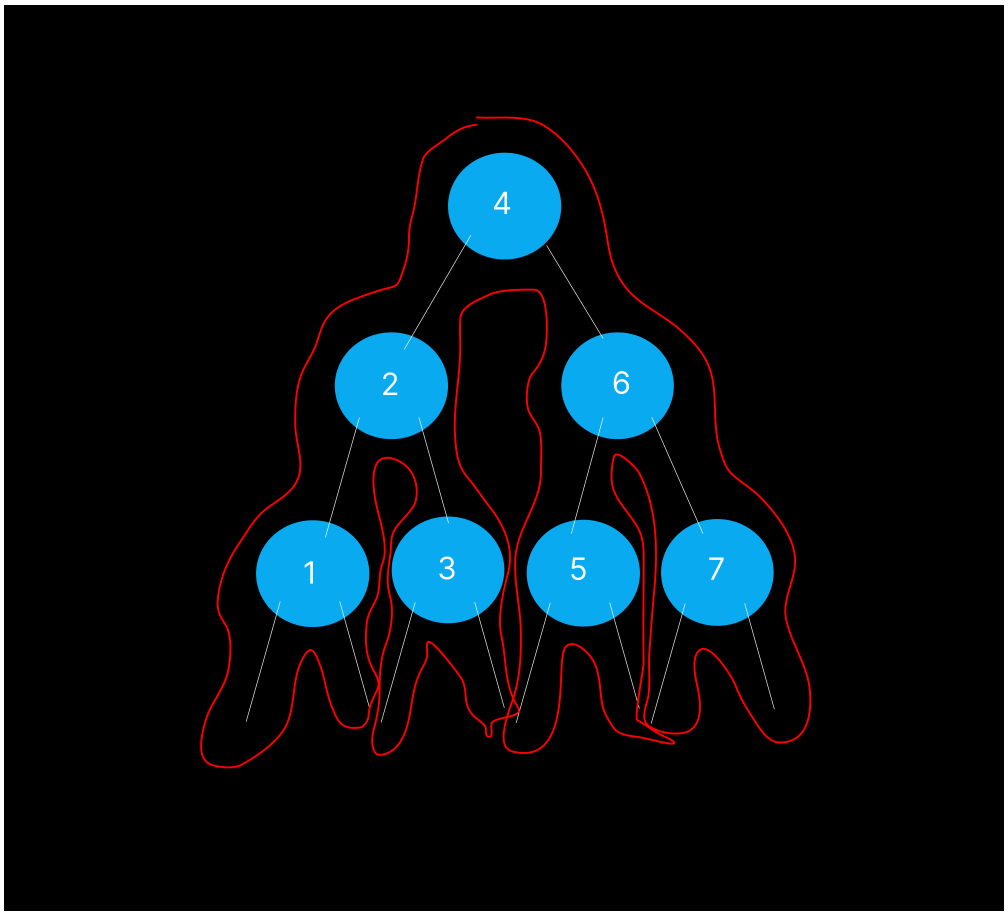
TreeSet Class -

- Null not allowed
- Duplicate not allowed
- Elements are stored in increasing order.
- Used Binary Search Tree data structure

Binary Search Tree:

- Parent node have zero, one and two nodes are allowed.
- Smaller elements on left side
- All elements are sorted.

Data: 4, 2, 3, 6, 5, 7, 1



Methods:

- All methods from SortedSet, Set, Collection and Iterable interface.
- heigher(E) ———- return the lowest element among all elements which is greater than specified element. Return null if not found
- lower(E) ———- return the highest element among all elements which are lower than specified elements. Return null if not found.
- ceiling(E) ———- return the lowest element among all elements which are grater than or equal to specified element.
- floor(E) ———- return the highest element among all elements which are less than or equal to specified elements.
- pollFirst() ——— return type E
- pollLast() ——— return type E
- headSet(Element, boolean) ——— return type NavigableSet<E>. Returns all the elements from zero index except specified element. Default flag is false. If we pass the boolean as true then it takes the specified element also.
- headSet(E) ——— return type SortedSet
- tailSet(E) ——— return type SortedSet
- tailSet(E,boolean) ——- return type NavigableSet<E>. Return the elements after the specified element. Default flag is true. If we pass the boolean value as false then it

will except the specified element.

- comparator()
- descendingIterator()
- descendingSet()

Map Interface -

- Everything is same as Set. Just difference is the values are stored in the form of key value pair format.
- We can use null values multiple times.
- One key can be null.
- Values can be stored duplicate but keys always unique.

Methods:

- put(key k, value v) ——— return type v
- putAll(map name) ——— return type void
- remove(object) ————— return type v
- clear() ————- return type void
- clone() ————- return type object
- entrySet() ——— return type Set
- containsKey(Object) ——— return type boolean
- containsValue(object) ————- return type boolean
- values() ——— return type collection.
- isEmpty() ——— return type boolean

Collections Class -

- Collections class in java is one of the utility class of java framework.
- All methods in collections class are static.

Methods:

- binarySearch(collection name, E)
- reverse()
- addAll()
- copy()
- replaceAll()
- shuffle()

- swap()
- rotate()
  And so on...

Java File Handeling, Java Multithreading, Java Serialization

This three topics we need to cover later due to words limit.