

Java

Core Java

(Simplified Text Book)



AMEERPET

T E C H N O L O G I E S

SRINIVAS GARAPATI

AMEERPET TECHNOLOGIES, NEAR SATYAM THEATRE, OPPOSITE HDFC BANK, AMEERPET, HYDERABAD
Contact for Online Classes: +91 – 911 955 6789 / 7306021113

Contents

S No	Topic	Page Num
01	Introduction to Java	02
02	Java Installation	04
03	Java API documentation	05
04	Java Application Structure	06
05	Naming Rules	07
06	Variables	08
07	Data types	09
08	Formatting output	13
09	Scanner class	15
10	Operators	17
11	Control Statements	26
12	OOPS – Introduction	38
13	Class Members	42
14	Static Members	44
15	Instance Members	50
16	Access Modifiers	54
17	Encapsulation	55
18	Inheritance	58
19	Polymorphism	63
20	Final Modifier	67
21	Abstraction	68
22	Interfaces	71
23	Objects – Relations (Use-A, Is-A, Has-A)	73
24	Factory & Singleton Classes	77
25	Arrays	79
26	Strings	84
27	Exception Handling	93
28	Multi-threading	103
29	IO Streams	109
30	Command Line Arguments	114
31	Packages	116

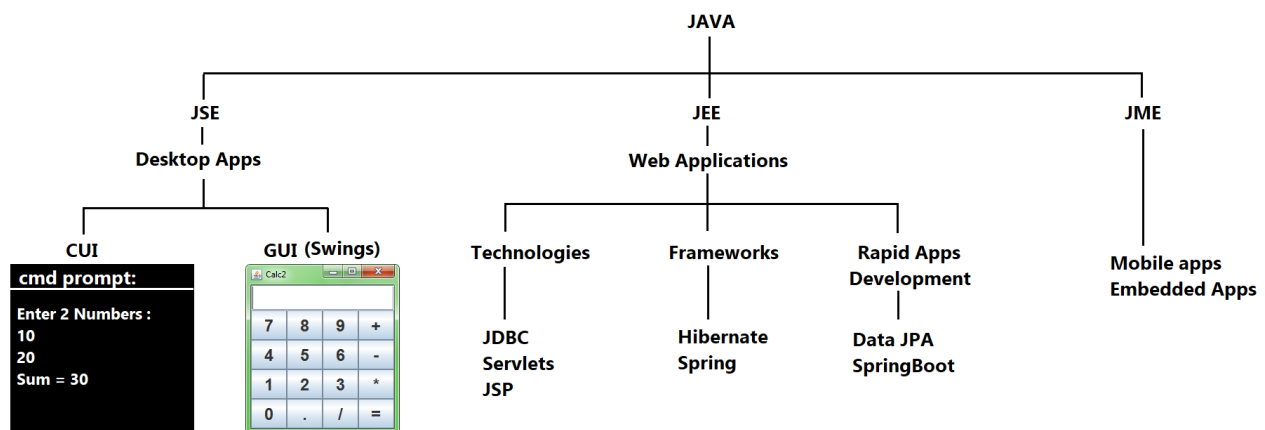
Core Java

Introduction:

- Java developed by James Gosling.
- Using Java, we can develop Desktop applications and Web applications.
- Using Core concepts of Java,
 - We develop Desktop applications.
- Using Java technologies (Servlets, JDBC and JSP)
 - We develop Web based applications.
- Using Frameworks (Spring, Hibernate, SpringBoot etc)
 - We achieve Rapid applications development.

Java distributions:

- **JSE(Java Standard Edition):** contains the core concepts of Java by which we can develop Desktop applications.
- **JEE(Java Enterprise Edition):** contains advanced concepts of Java to develop Server side applications.
- **JME(Java Micro Edition):** concepts used to develop mobile applications, embedded systems etc.



Java is a platform independent language:

- Java Compiler converts the Source program into Byte code format file.
- Byte code can run on any OS, hence java applications become platform independent.

Java is an Object-Oriented language:

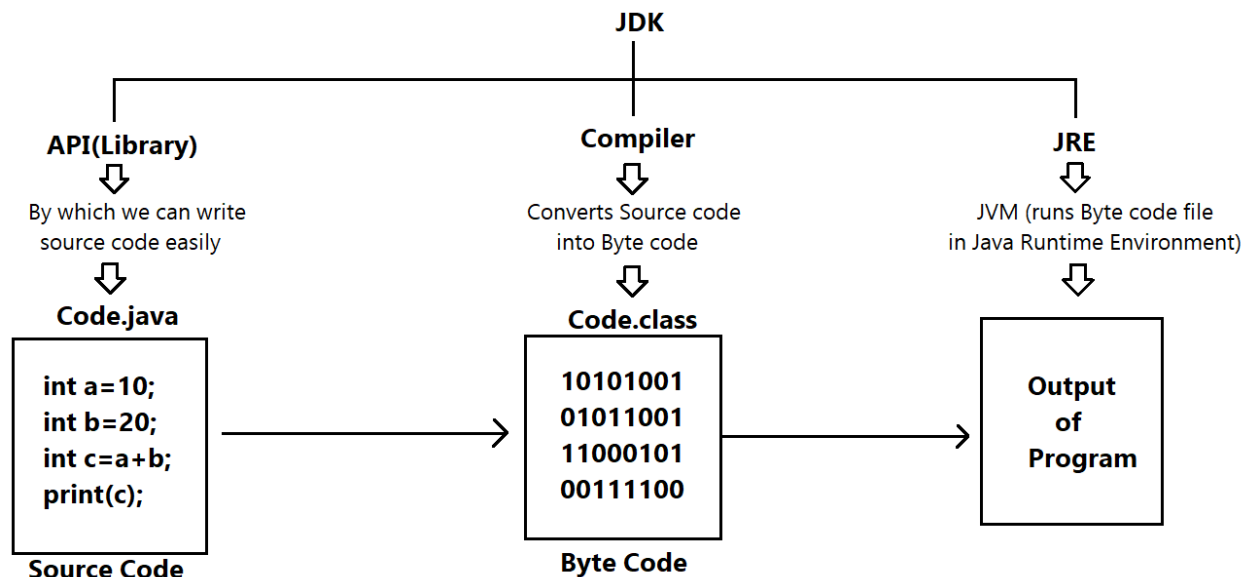
- Object oriented programming is a way of organizing programs as collection of objects, each of which represents an instance of a class.
- **Four main concepts of OOPS:**
 - Abstraction
 - Encapsulation
 - Inheritance
 - Polymorphism

Multithreading:

- Java supports multithreading.
- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

JDK Components to Edit, Compile and Run Java Application:

1. **API**(Application Program Interface): A Library of programs by which we can develop java application easily
2. **Compiler**(javac): converts source code (.java file) to the byte code(.class file).
3. **JRE**(Java Runtime Environment): An environment set up in which Java application runs.
4. **JVM**(Java Virtual Machine): JVM runs Java Application from JRE.



IDE: (Integrated Development Environment)

- A software by which we can develop and run java applications easily
- IDE provides intelligence that helps the developer.
- Java Basic IDEs – Notepad++ , EditPlus
- Java Realtime IDEs – Eclipse, NetBeans, IntelliJ, STS etc.

Java - Installation

Installing JDK:

- We can download the latest version from official website:
<https://www.oracle.com/java/technologies/downloads/>
- Download based on your Operating System.
- For windows, download executable(.exe) file

Click on run button once download completes:

The JDK includes tools for developing and testing programs written in the Java programming language and running on the Java platform.

Linux

macOS

Windows

Product/file description	File size	Download
x64 Compressed Archive	179.05 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.zip (sha256)
x64 Installer	158.84 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.exe (sha256)
x64 MSI Installer	157.70 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.msi (sha256)

- After installation of JDK, we use two commands from installed path.
 - javac(java compiler)
 - java (java interpreter)
- Path is : C:\Program Files\Java\jdk-17.0.4.1\bin

This PC > Windows (C:) > Program Files > Java > jdk-17.0.4.1 > bin

Name	Date modified	Type	Size
jaccessinspector	9/18/2022 11:14 AM	Application	104 KB
jaccesswalker	9/18/2022 11:14 AM	Application	70 KB
jar	9/18/2022 11:14 AM	Application	24 KB
jarsigner	9/18/2022 11:14 AM	Application	24 KB
java.dll	9/18/2022 11:14 AM	Application extens...	144 KB
java	9/18/2022 11:14 AM	Application	54 KB
javaaccessbridge.dll	9/18/2022 11:14 AM	Application extens...	282 KB
javac	9/18/2022 11:14 AM	Application	24 KB
javadoc	9/18/2022 11:14 AM	Application	24 KB
javajpeg.dll	9/18/2022 11:14 AM	Application extens...	175 KB

← JVM

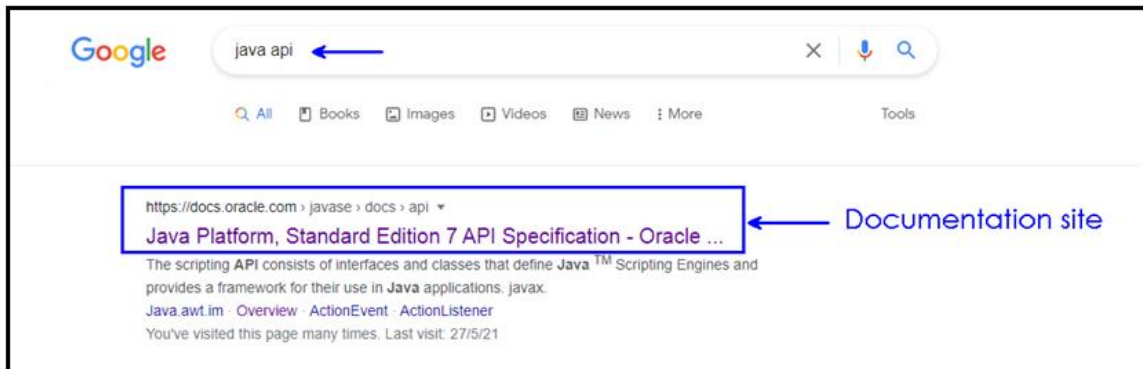
← Compiler

Java API Documentation

Java API Documentation:

- Java Library is a pre-defined collection of packages, classes, interface etc.
- Library is technically called "Java API"
- The documentation (theoretical explanation) of every program available as API documentation.
- Oracle corporation providing API documentation as official website.
- <https://docs.oracle.com/javase/7/docs/api/> is giving information about library.

Search "java api" in Google:



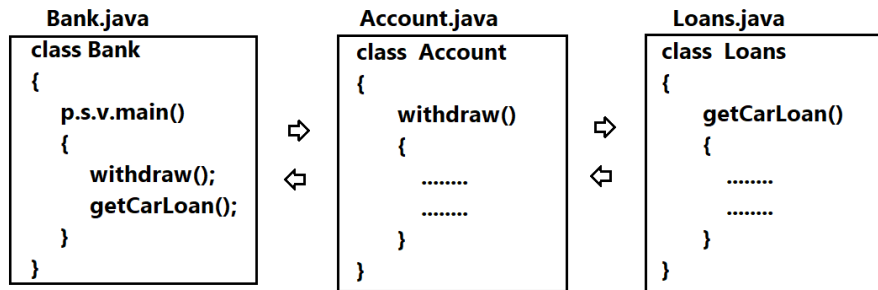
The Website providing complete information as follows:

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.

Java Application Structure

Java application structure:

- Java application is a collection of Source files.
- Source file consists one or more classes.
- Class contains Variables and Methods.
- Java application runs by JVM.
- JVM invoke main() method to start application execution.
- Application starts with one class in which main() method has defined.



How to Compile and Run Java program from command Prompt?

1. Open Notepad, Write Code and Save file with .java extension.
2. Open Command prompt(CUI based OS).
3. Go to file location.
4. Compile using command – javac
5. Run using - java

Code.java:

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

Command Prompt as follows:

```
C:\Users\G Srinivas>cd\
C:\>d:
D:\>cd practice
D:\practice>javac Code.java
D:\practice>java Code
Hello world!
```

Naming Rules in Java

Naming conventions:

- It is recommended to follow Java Naming rules while naming classes, methods, variables, interfaces, constants etc.
- Java Source code become more readable and understandable with naming rules.

Class: Every word in identity starts with capital letter. Spaces not allowed

```
String  
PrintStream  
NullPointerException  
FileNotFoundException  
ArrayIndexOutOfBoundsException
```

Method: First word starts with Lower Case and From Second word, every word starts with capital letters. No spaces allowed.

```
main( )  
getName( )  
getAccountHolderNumber( )
```

Variable: First word starts with Lower Case and From Second word, every word starts with capital letters. No spaces allowed

```
int sum = 0 ;  
float shoeSize = 8.9 ;
```

Constant variable : All letters should be in upper case, words separate with underscore(_)

```
MIN_VALUE = 0;  
MAX_PRIORITY = 10;
```

Package: package must represent with lower case letters.

```
java, lang, io, util, sql, http, servlet...
```

Abstract Class and Interface: Same as class

```
Runnable (interface)  
InputStream (abstract class)
```


Variables

Variable: Identity given to memory location.

or

Named memory location

Syntax:

```
datatype identity = value;
```

Examples:

```
String name = "Amar";  
int age = 23;  
char gender = 'M';  
boolean married = false;  
double salary = 35000.00;
```

Declaration, Assignment, Update and Initialization of Variable:

1. Declaration: creating a variable without any value.

```
int a ;
```
2. Assignment: Storing a value into variable which is already declared.

```
a = 10 ;
```
3. Modify: Increase or Decrease the value of variable.

```
a = a + 5 ;
```
4. Initialization: Assigning value at the time of declaration

```
int b = 20 ;
```

Identifier rules:

- Identifier is a name given to class, method, variable etc.
- Identifier must be unique
- Identifier is used to access the member.

Rules to create identifier:

1. Identifier allows [A-Z] or [a-z] or [0-9], and underscore(_) or a dollar sign (\$).
2. Spaces not allowed in identifier.
3. Identifier should not start with digit.
4. Identifier can start with special symbol.
5. We can't use the Java reserved keywords as an identifier.

Java - Datatypes

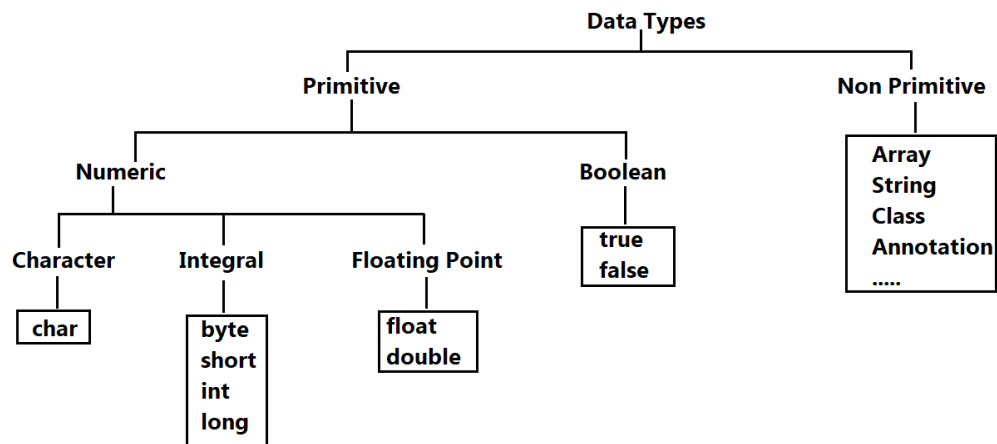
Data type:

- In the declaration of every variable, it is mandatory to specify its data type.
- Data type describes,
 1. The size of memory allocated to variable.
 2. The type of data allowed to store into variable.

Syntax: data_type variable_name = value ;

Example: int sum = 0;
 float balance = 3000.0;
 String name = "Amar";

Data types classified into Primitive and Non-Primitive types.



Limits:

- A data type limit describes the minimum and maximum values we can store into a variable.
- Every data type is having size and it is represented by bytes
- One byte equals to 8 bits.

Integer types: We have 4 integer types among eight available primitive types in java.

Type	Size(bytes)	Limits
byte	1	-128(-2 ⁷) to +127(2 ⁷ -1)
short	2	-32,768(-2 ¹⁵) to 32,767(2 ¹⁵ -1)
int	4	-2,147,483,648(-2 ³¹) to 2,147,483,647(2 ³¹ -1)
long	8	-9,223,372,036,854,775,808(-2 ⁶³) to 9,223,372,036,854,775,807(2 ⁶³ -1)

Note:

- For every primitive type, there is a corresponding wrapper class.
- Wrapper classes providing pre-defined variables to find Size and Limits.

Program to print sizes of Integer types:

```
class Sizes {  
    public static void main(String[] args) {  
        System.out.println("Byte size : " + Byte.SIZE/8 + " bytes");  
        System.out.println("Short size : " + Short.SIZE/8 + " bytes");  
        System.out.println("Integer size : " + Integer.SIZE/8 + " bytes");  
        System.out.println("Long size : " + Long.SIZE/8 + " bytes");  
    }  
}
```

Program to print Limits of Integer types:

```
class Limits {  
    public static void main(String[] args) {  
        System.out.println("Byte min val : " + Byte.MIN_VALUE);  
        System.out.println("Byte max val : " + Byte.MAX_VALUE);  
        System.out.println("Short min val : " + Short.MIN_VALUE);  
        System.out.println("Short max val : " + Short.MAX_VALUE);  
    }  
}
```

Character data type: Character type variable is used to store alphabets, digits and symbols.

```
class Code{  
    public static void main(String[] args) {  
        char x = 'A';  
        char y = '5';  
        char z = '$';  
        System.out.println("x val : " + x + "\ny val : " + y + "\nz val : " + z);  
    }  
}
```

Note: We must represent Character with Single Quotes in Java

ASCII : Representation of each character of language using constant integer value.

A-65	a-97	0-48	*-34
B-66	b-98	1-49	#-35
...	\$-36
...
...
Z-90	z-122	9-57	...

Type casting:

- Conversion of data from one data type to another data type.
- Type casting classified into
 - **Implicit cast:** Auto conversion of data from one type to another type.
 - **Explicit cast:** Manual conversion of data from one type to another type.

Convert Character to Integer: this conversion happens implicitly when we assign character to integer type.

```
class Code
{
    public static void main(String[] args) {
        char ch = 'A' ;
        int x = ch ;
        System.out.println("x value : " + x);
    }
}
```

Convert Integer to Character: this conversion must be done manually

```
class Code
{
    public static void main(String[] args) {
        int x = 97 ;
        char ch = (char)x ;
        System.out.println("ch value : "+ch);
    }
}
```

Convert Upper case to Lower case Character:

Upper Case		Lower Case
A = 65	→ +32	a = 97
B = 66		b = 98
..	← -32	..
..		..
Z = 90		z = 122

```

class Code {
    public static void main(String[] args) {
        char up = 'A';
        char lw = (char)(up+32);
        System.out.println("Result : " + lw);
    }
}

```

Convert Digit to Number: We can convert character type digit into number as follows

Digits ASCII	Number
'0' = 48	'0' - 48 --> 48-48 = 0
'1' = 49	'1' - 48 --> 49-48 = 1
..	..
..	..
'9' = 57	'9' - 48 --> 57-48 = 9

```

class Code {
    public static void main(String[] args) {
        char d = '5';
        int n = d-48;
        System.out.println("Result : " + n);
    }
}

```

Note: Type casting not required in above code as we are assigning value to integer variable.

Decimal Numbers:

- We store decimal values using 2 types float and double.

float type	Size in bytes	Limits
float	4	$\pm 3.40282347\text{E}+38\text{F}$
double	8	$\pm 1.79769313486231570\text{E}+308$

- We must specify the float type with precision f or F
- By default, double precision d or D will be added at end.

```

class Code {
    public static void main(String[] args) {
        float f = 2.3f;
        System.out.println("f value : "+f);
    }
}

```

Formatting Output

Introduction: It is important to format the output before display the results to end user. Proper formatting makes the user more understandable about program results.

We always display results in String format. To format the output, we concatenate the values such as int, char, double, boolean with messages (string type) as follows:

Syntax	Example
int + int = int	10 + 20 = 30
String + String = String	"Java" + "Book" = JavaBook "10" + "20" = 1020
String + int = String	"Book" + 1 = Book1 "123" + 456 = 123456
String + int + int = String	"Sum = " + 10 + 20 = Sum1020
String + (int + int) = String	"Sum = " + (10 + 20) = Sum30
String + double = String	"Value = " + 23.45 = Value = 23.45
String – int = Error	

```
public class Code
{
    public static void main(String[] args) {
        int a=10;
        System.out.println(a);
    }
}
```

Output: 10

```
public class Code
{
    public static void main(String[] args) {
        int a=10, b=20;
        System.out.println(a + " , " + b);
    }
}
```

Output: 10, 20

```
public class Code
{
    public static void main(String[] args) {
        int a=10, b=20;
        System.out.println("a = " + a);
    }
}
```

```
        System.out.println("b = " + b);
    }
}
```

Output: **a = 10**
 b = 20

```
public class Code
{
    public static void main(String[] args) {
        int a=10, b=20;
        System.out.println("a = " + a + " , " + "b = " + b);
    }
}
```

Output: **a = 10, b = 20**

```
public class Code
{
    public static void main(String[] args) {
        int a=10, b=20;
        int c=a+b;
        System.out.println("Sum of " + a + " and " + b + " is " + c);
    }
}
```

Output: Sum of 10 and 20 is 30

```
public class Code
{
    public static void main(String[] args) {
        int a=10, b=20;
        int c=a+b;
        System.out.println(a + " + " + b + " = " + c);
    }
}
```

Output: 10+20=30

Reading input - Scanner Class

Scanner Class:

- Using java library class Scanner, we can read input like integers, characters, strings, double values from the user.
- Different methods to read different values such as nextInt(), next(), nextDouble()...
- We need to specify the Keyboard(System.in) while creating Scanner class object.
 - Scanner scan = new Scanner(System.in);
- We access all the methods using object reference name.

Reading integer value: nextInt() method read and returns an integer value

```
import java.util.Scanner;
class ReadInt
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter integer : ");
        int n = scan.nextInt();
        System.out.println("n value is : " + n);
    }
}
```

Output: Enter integer : 10
n value is : 10

Reading Boolean value: nextBoolean() method returns Boolean value that we entered

```
import java.util.Scanner;
class ReadBoolean
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter boolean value : ");
        boolean b = scan.nextBoolean();
        System.out.println("b value is : " + b);
    }
}
```

Output: Enter Boolean value : true
b value is : true

Reading String: using next() method we can read single word string from the user

```
import java.util.Scanner;
```



```
class Code
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter your name : ");
        String name = scan.next();
        System.out.println("Hello : " + name);
    }
}
```

Output: Enter your name: Amar
Hello : Amar

Reading character: There is no method in Scanner class to read single character, hence we read first character of String to read single character as follows

```
import java.util.Scanner;
class Code {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter character : ");
        char ch = scan.next().charAt(0);
        System.out.println("Input character is : " + ch);
    }
}
```

Output: Enter character : A
Input character is : A

Reading multi-word String: nextLine() method can read string includes spaces

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter string with spaces : ");
        String s = scan.nextLine();
        System.out.println("Input String is : " + s);
    }
}
```

Output : Enter string with spaces : This is core concept
Input String is : This is core concept

Java - Operators

Operator: Operator is a symbol that performs operation on operands.

Arithmetic operators:

- These operators perform all arithmetic operations.
- Operators are +, -, *, /, %
- Division(/) : returns quotient after division
- Mod(%) : returns remainder after division

```
public class Code {
    public static void main(String[] args) {
        int a=8, b=5 ;
        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));
    }
}
```

Operators Priority: We need to consider the priority of operators if the expression has more than one operator. Arithmetic operators follow BODMAS rule.

Priority	Operator
()	First. If nested then inner most is first
*, / and %	Next to (). If several, from left to right
+, -	Next to *, /, %. If several, from left to right

Examples expressions with evaluations:

5+3-2	5*3%2	5+3*2	(5+3)*2
8-2	15%2	5+6	8*2
6	1	11	16

Practice codes on Arithmetic operators

Find the average of two numbers:

```
class Code
{
    public static void main(String[] args) {
        int num1= 5 , num2 = 3;
        int avg = (num1+num2)/2;
        System.out.println("Average : " + avg);
    }
}
```

Print Last Digit of given Number:

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a number : ");
        int n = scan.nextInt();
        int d = n%10;
        System.out.println("Last digit of " + n + " is " + d);
    }
}
```

Remove Last Digit of Given Number

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a number : ");
        int n = scan.nextInt();
        n = n/10;
        System.out.println("After remove last digit : " + n);
    }
}
```

Calculate Total Salary of Employee:

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter basic salary : ");
        double basic = scan.nextDouble();
        double hra = 0.25*basic;
        double ta = 0.2*basic;
        double da = 0.15*basic;
        double total = basic + hra + ta + da ;
        System.out.println("Total Salary : " + total);
    }
}
```

Relational operators: These operator return boolean value by validating the relation on data.

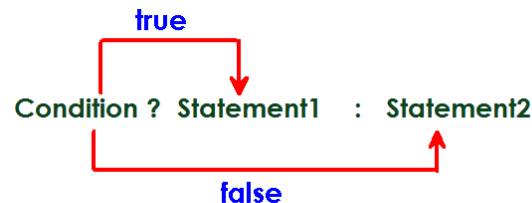
- Operators are >, <, >=, <=, ==, !=

class Code {

```
    public static void main(String[] args) {  
        int a=5, b=3 ;  
        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));  
    }  
}
```

Conditional Operator:

- It is widely used to execute condition in true part or in false part.
- It is a simplified form of if-else control statement.



Program to Find the Biggest of 2 numbers:

```
class Code{  
    public static void main(String[] args) {  
        int a=5,b=2,c=0 ;  
        c = a > b ? a : b ;  
        System.out.println("Big : "+c);  
    }  
}
```

Program to check the given number is Even or Not:

```
class EvenOrNot {  
    public static void main(String[] args) {  
        int a=5 ;  
        boolean res ;  
        res = a%2==0 ? true : false ;  
        System.out.println(a + " is Even : "+res);  
    }  
}
```

Modify operators:

- These are also called Increment and Decrement operators.
- Modify operators are increase and decrease the value of variable by 1.
- Operators are ++, -- ;

Increase value of a variable by 1 in 3 ways:

x=10	->	x=x+1	->	x=11
x=10	->	x+=1	->	x=11
x=10	->	x++	->	x=11

Decrease value of a variable by 1 in the same way:

x=10	->	x=x-1	->	x=9
x=10	->	x-=1	->	x=9
x=10	->	x--	->	x=9

Logical Operators

- A logical operator is primarily used to evaluate multiple expressions.
- Logical operators return boolean value.
- Operators are && , || and !

When 'A' is	When 'B' is	A&&B is	A B is	!A is	!B is
T	T	T	T	F	F
F	T	F	T	T	F
T	F	F	T	F	T
F	F	F	F	T	T

```
class Logical {
    public static void main(String[] args) {
        System.out.println("true && true : " + (true && true));
        System.out.println("true && false : " + (true && false));
        System.out.println("true || false : " + (true || false));
        System.out.println("false || false : " + (false || false));
    }
}
```

```
class Logical {
    public static void main(String[] args) {
        System.out.println("!true : " + (!true));
        System.out.println("!false : " + (!false));
    }
}
```

Bitwise Operators:

- These operators are used to process the data at bit level.
- These can be applied only integers and characters.
- Operators are &, |, ^.
- These operators process the data according to truth table.

a	b	a&b	a b	a^b	~a	~b
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

```

class Bitwise{
    public static void main(String[] args) {
        int a=10, b=8 , c, d, e;
        c = a&b ;
        d = a|b ;
        e = a^b ;
        System.out.println(c+","+d+","+e);
    }
}

```

Shift Operators:

- The bitwise shift operators move the bit values of a binary object.
- The left operand specifies the value to be shifted.
- The right operand specifies the number of positions that the bits in the value are to be shifted.
- The bit shift operators take two arguments, and looks like:
 - x << n
 - x >> n

```

class Shift{
    public static void main(String[] args) {
        int a=8, b, c;
        b = a>>2 ;
        c = a<<2 ;
        System.out.println(a+","+b+","+c);
    }
}

```

Conditions using Arithmetic, Relational and Logical operators

Condition to check the number is Positive or Negative:

Variables:
 int a=?;

Condition:
 a >= 0

Condition to check the number is equal to zero or not:

Variables:
 int a=?;

Condition:
 a == 0

Condition to check the 2 numbers equal or not:

Variables:
 int a=?, b=?;

Condition:
 a == b

Condition to check First Num greater than Second Num or Not:

Variables:
 int a=?, b=?;

Condition:
 a > b

Square of First Number equals to Second Number or Not :

Variables:
 int a=?, b=?;

Condition:
 a*a == b

Condition to check sum of 2 numbers equal to 10 or not:

Variables:
 int a=?, b=? ;

Condition:
 a+b == 10 ;

Condition to check the number divisible by 3 or not:

Variables:
 int n=?;

Condition:
 n % 3 == 0

Condition to check the number is Even or not:

Variables:
 int n=?;

Condition:
 n % 2 == 0

Condition to check the last digit of Number is 0 or not:

Variables:
 int n=?;

Condition:
 n % 10 == 0

Condition to check the multiplication of 2 numbers not equals to 3rd number or not:

Variables:
 int a=?, b=?, c=?;

Condition:
 a * b != c

Condition to check average of 4 subjects marks greater than 60 or not:

Variables:
 int m1=?, m2=?, m3=? , m4=? ;

Condition:
 (m1 + m2+ m3+ m4)/4 >= 60

Condition to check the sum of First 2 numbers equals to last digit of 3rd num or not:

Variables:
 int a=?, b=? , c=? ;

Condition:
 a + b == c% 10

Condition to check the given quantity of fruits exactly in dozens or not:

Variables:
 int quantity = ?;

Condition:
 quantity%12 == 0

Condition to person is eligible for Vote or Not:

Variables:
 int age=?;

Condition:
 age >= 18

Condition to check First Num is greater than both Second & Third Nums or Not:

Variables:
 int a=?, b=?, c=?;

Condition:
 a > b && a > c

Condition to check the number divisible by both 3 and 5 or not:

Variables:
 int n=?;

Condition:
 n % 3 == 0 && n % 5 == 0

Condition to check the number is in between 30 and 50 or not:

Variables:
 int n=?;

Condition:
 n >= 30 && n <= 50

Condition to check the student passed in all 5 subjects or Not:

Variables:
 int m1=?, m2=?, m3=?, m4=?, m5=?;

Condition:
 m1>=35 && m2>=35 && m3>=35 && m4>=35 && m5>=35

Condition to check the character is Vowel or Not:

Variables:

```
int ch = '?';
```

Condition:

```
ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u'
```

Condition to check the character is Upper case alphabet or not:

Variables:

```
int ch='?';
```

Condition:

```
ch >= 'A' && ch <= 'Z'
```

Condition to check the character is Alphabet or not:

Variables:

```
int ch='?';
```

Condition:

```
(ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z')
```

Condition to check the 3 numbers equal or not:

Variables:

```
int a=?, b=?, c=?;
```

Condition:

```
a == b && b == c
```

Condition to check any 2 numbers equal or not among the 3 numbers:

Variables:

```
int a=? , b=? , c=?;
```

Condition:

```
a==b || b==c || c==a
```

Condition to check the 3 numbers are unique or not:

Variables:

```
int a=? , b=? , c=?;
```

Condition:

```
a!=b && b!=c && c!=a
```

Control Statements

Sequential statements:

- Statement is a line of code.
- Sequential Statements execute one by one from top-down approach.

Control statements: Statements execute randomly and repeatedly based on conditions.

Conditional	Loop	Branching
If-block	For loop	Break
If-else block	While loop	Continue
If-else-if block	Do-while loop	Return
Nested if block	Nested loops	
Switch case		

If block: execute a block of instructions only when condition is valid.

Syntax	Flow Chart
<pre>if(condition) { Logic; }</pre>	<pre> graph TD Start([Start]) --> Condition{Condition} Condition --> Logic[/If-block-logic/] Logic --> End([End]) Condition --> End </pre>

Example: Give 10% discount if the bill amount is > 5000

```
class Code
{
    public static void main(String[] args)
    {
        double bill = 6000;
        if(bill > 5000){
            double discount = 0.1*bill ;
            bill = bill - discount;
        }
        System.out.println("Total bill amount is : " + bill);
    }
}
```

if - else block: Else block can be used to execute an optional logic if the given condition has failed.

Syntax	Flow Chart
<pre> if (condition){ If – Stats; } else{ Else – Stats; } </pre>	<pre> graph TD Start([Start]) --> Condition{Condition} Condition -- True --> IfStats[If - Stats] Condition -- False --> ElseStats[Else - stats] IfStats --> End([End]) ElseStats --> End </pre>

Program to check the number is even or odd

Even Number: The number which is divisible by 2

class Code

```

{
    public static void main(String[] args) {
        int n=5;
        if(n%2==0)
            System.out.println("Even number");
        else
            System.out.println("Not even number");
    }
}

```

Program to Check Person can Vote or Not:

import java.util.Scanner;

class Code

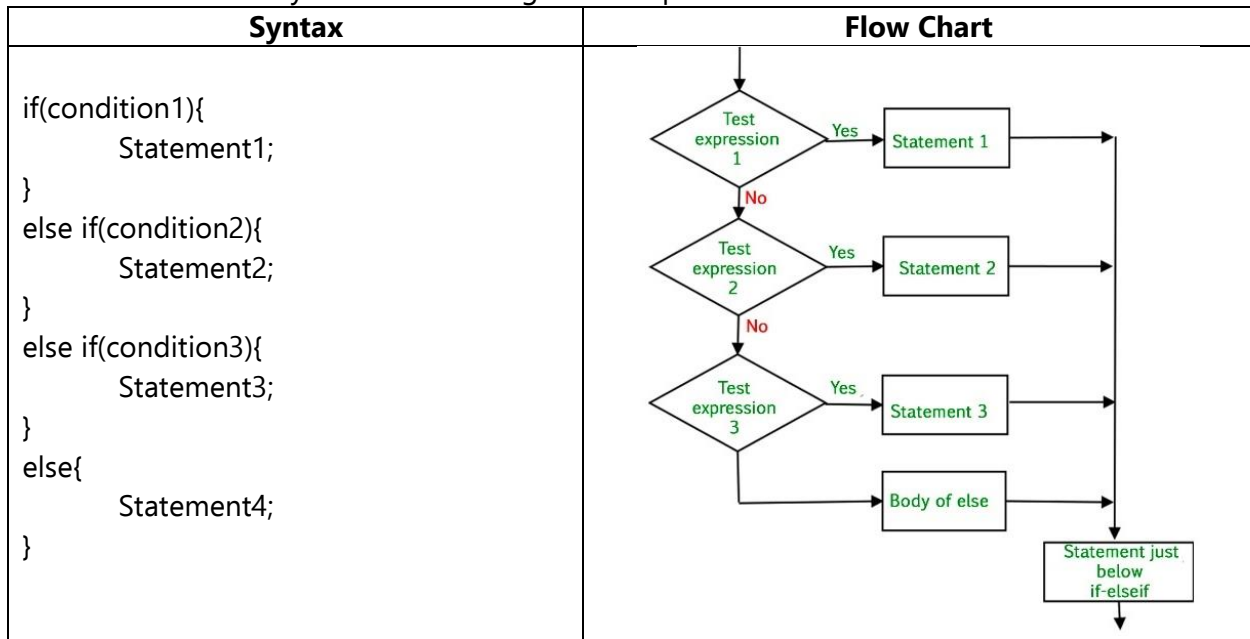
```

{
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter age : ");
        int age = scan.nextInt();
        if(age>=18)
            System.out.println("Eligible for Vote");
        else
            System.out.println("Wait " + (18-age) + " more years to vote");
    }
}

```

If-else-if ladder:

- It is allowed to define multiple if blocks sequentially.
- It executes only one block among the multiple blocks defined.

**Program to find Biggest of Three Numbers:**

```

import java.util.Scanner;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a, b, c values : ");
        int a = scan.nextInt();
        int b = scan.nextInt();
        int c = scan.nextInt();
        if(a>b && a>c)
            System.out.println("a is big");
        else if(b>c)
            System.out.println("b is big");
        else
            System.out.println("c is big");
    }
}

```

Program to check the Character is Alphabet or Digit or Symbol

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter character : ");
        char ch = scan.nextLine().charAt(0);
        if((ch>='A' && ch<='Z') || (ch>='a' && ch<='z'))
            System.out.println("Alphabet");
        else if(ch>='0' && ch<='9')
            System.out.println("Digit");
        else
            System.out.println("Special Symbol");
    }
}
```

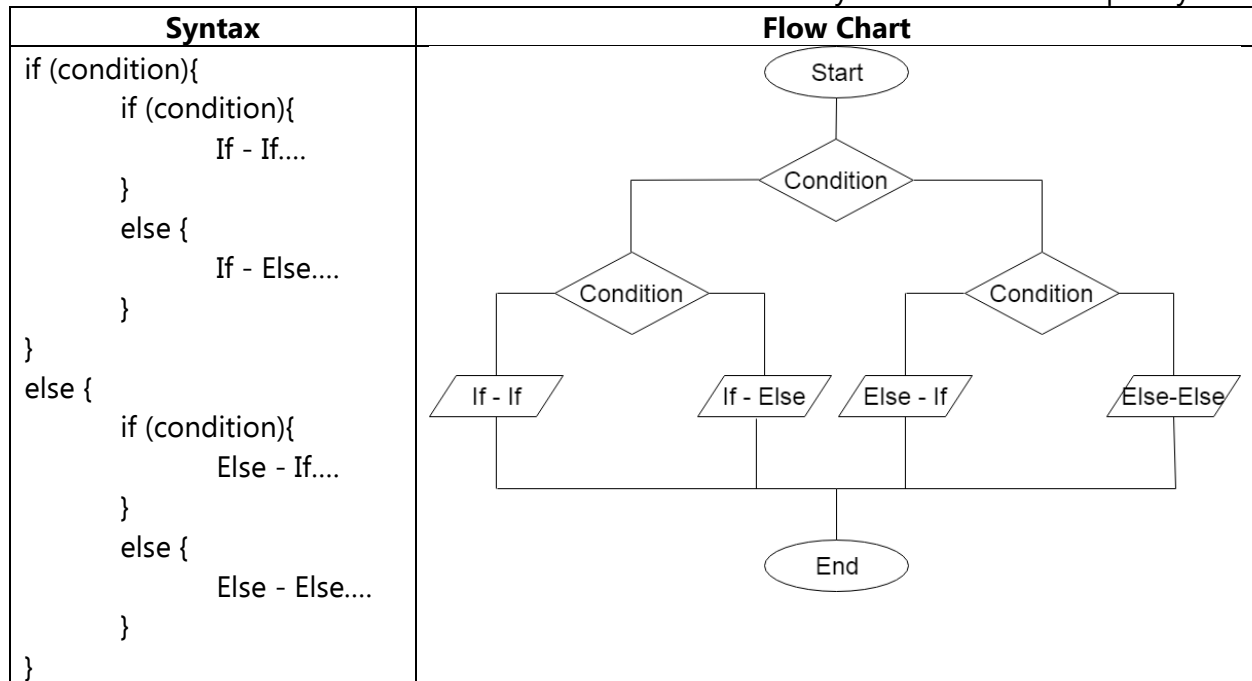
Program to check the input year is Leap or Not:

Leap Year rules:

- 400 multiples are leap years : 400, 800, 1200, 1600....
- 4 multiples are leap years: 4, 8, 12, ... 92, 96...
- 100 multiples are not leap years: 100, 200, 300, 500, 600...

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter year : ");
        int n = scan.nextInt();
        if((n%400==0))
            System.out.println("Leap year");
        else if(n%4==0 && n%100!=0)
            System.out.println("Leap Year");
        else
            System.out.println("Not leap year");
    }
}
```

Nested if block: Defining a block inside another block. Inner if block condition evaluates only if the outer condition is valid. It is not recommended to write many levels to avoid complexity.



Give the student grade only if the student passed in all subjects:

class Code

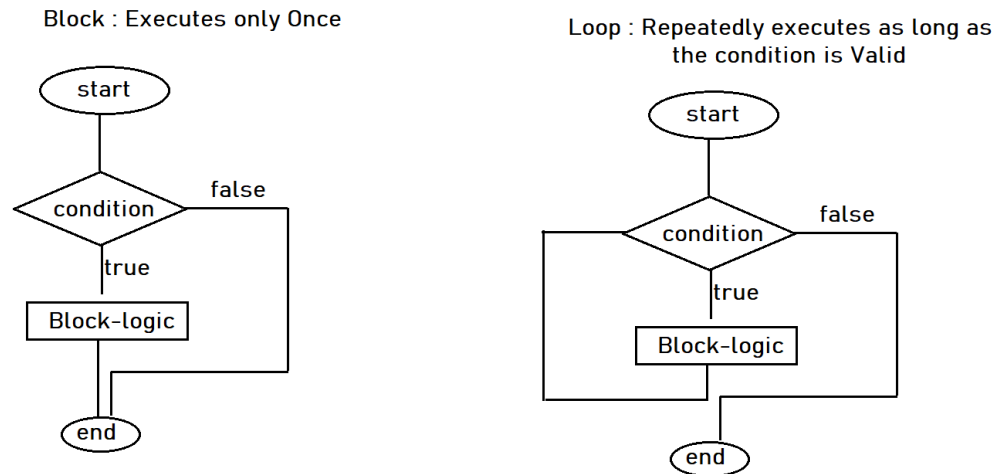
```

{
    public static void main(String[] args) {
        int m1=45, m2=67, m3=44;
        if(m1>=40 && m2>=40 && m3>=40){
            int total = m1+m2+m3;
            int avg = total/3;
            if(avg>=60)
                System.out.println("Grade-A");
            else if(avg>=50)
                System.out.println("Grade-B");
            else
                System.out.println("Grade-C");
        }
        else
        {
            System.out.println("Student failed");
        }
    }
}

```

Introduction to Loops

Loop: A Block of instructions execute repeatedly as long the condition is valid.



Note: Block executes only once whereas Loop executes until condition become False

Java Supports 3 types of Loops:

1. For Loop
2. While Loop
3. Do-While Loop

For Loop: We use for loop only when we know the number of repetitions. For example,

- Print 1 to 10 numbers
- Print Array elements
- Print Multiplication table
- Print String character by character in reverse order

While loop: We use while loop when we don't know the number of repetitions.

- Display contents of File
- Display records of Database table

Do while Loop: Execute a block at least once and repeat based on the condition.

- ATM transaction: When we swipe the ATM card, it starts the first transaction. Once the first transaction has been completed, it asks the customer to continue with another transaction and quit.

For Loop

for loop: Execute a block of instructions repeatedly as long as the condition is valid. We use for loop only when we know the number of iterations to do.

Syntax	Flow Chart
<pre>for(init ; condition ; modify) { statements; } for(start ; stop ; step) { statements; }</pre>	<pre> graph TD Start([Start]) --> Condition{Condition} Condition -- FALSE --> End([End]) Condition -- TRUE --> Statements[Statements] Statements --> Condition </pre>

Program to Print 1 to 10 Numbers

```
class Code {
    public static void main(String[] args) {
        for (int i=1 ; i<=10 ; i++){
            System.out.println("i val : " + i);
        }
    }
}
```

Program to Print 10 to 1 numbers

```
class Code{
    public static void main(String[] args) {
        for (int i=10 ; i>=1 ; i--){
            System.out.println("i val : " + i);
        }
    }
}
```

Program to display A-Z alphabets

```
class Code {
    public static void main(String[] args) {
        for (char ch='A' ; ch<='Z' ; ch++){
            System.out.print(ch + " ");
        }
    }
}
```

While Loop

While loop: Execute a block of instructions repeatedly until the condition is false. We use while loop only when don't know the number of iterations to do.

Syntax	Flow Chart
<pre>while(condition) { statements; }</pre>	<pre> graph TD Start([Start]) --> Condition{Condition?} Condition -- True --> DoTask[Do Task] DoTask --> Condition Condition -- False --> End([End]) </pre>

Program to Count the digits in given number

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter Num : ");
        int n = scan.nextInt();
        int count=0;
        while(n!=0)
        {
            n=n/10;
            count++;
        }
        System.out.println("Digits count : " + count);
    }
}
```

Display sum of the digits in the given number

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter Num : ");
```

```
int n = scan.nextInt();
int sum=0;
while(n!=0)
{
    sum = sum + n%10;
    n=n/10;
}
System.out.println("Sum of digits : " + sum);
}
```

Check the given 3 digit number is Armstrong Number or not :

Sum of cubes of individual digits equals to the same number

Example: 153 -> $1^3 + 5^3 + 3^3 = 153$

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter Num : ");
        int n = scan.nextInt();
        int temp, sum=0, r;
        temp=n;
        while(n>0)
        {
            r = n%10;
            sum = sum + r*r*r;
            n = n/10;
        }
        if(temp==sum)
            System.out.println("ArmStrong Number");
        else
            System.out.println("Not an ArmStrong Number");
    }
}
```

Break and Continue

break: A branching statement that terminates the execution flow of a Loop or Switch case.

```
class Code {  
    public static void main(String[] args) {  
        for (int i=1 ; i<=10 ; i++){  
            if(i==5){  
                break;  
            }  
            System.out.print(i + " ");  
        }  
    }  
}
```

Output: 1 2 3 4

Break statement terminates the flow of infinite loop also:

```
class Code {  
    public static void main(String[] args) {  
        while(true){  
            System.out.print("Loop");  
            break;  
        }  
    }  
}
```

Output: Loop prints infinite times

Continue: A branching statement that terminates the current iteration of loop execution.

```
class Code {  
    public static void main(String[] args) {  
        for (int i=1 ; i<=10 ; i++){  
            if(i==5){  
                continue;  
            }  
            System.out.print(i + " ");  
        }  
    }  
}
```

Output: 1 2 3 4 5 6 7 8 9 10

Switch case

Switch: It is a conditional statement that executes specific set of statements(case) based on given choice. Default case executes if the user entered invalid choice. Case should terminate with break statement.

Syntax	FlowChart
<pre> switch(choice) { case 1 : Statements ; break case 2 : Statements ; break case n : Statements ; break default: Statements ; } </pre>	<pre> graph TD Start(()) --> Expression{expression} Expression --> Case1{case-1} Case1 -- Matched --> S1[Statement-1] S1 --> B1[break] B1 --> Exit(()) Case1 -- Unmatched --> Case2{case-2} Case2 -- Matched --> S2[Statement-2] S2 --> B2[break] B2 --> Exit Case2 -- Unmatched --> CaseN{case-n} CaseN -- Matched --> SN[Statement-n] SN --> BN[break] BN --> Exit CaseN -- Unmatched --> Default{default} Default --> SS[Statement-S] SS --> BS[break] BS --> Exit </pre>

```
import java.util.Scanner;
```

```
class Code {
```

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

```
        Scanner sc = new Scanner(System.in);
```

```
        System.out.print("Enter character(r, g, b) : ");
```

```
        char ch = sc.next().charAt(0);
```

```
        switch(ch){
```

```
            case 'r' :    System.out.println("Red");
                        break;
```

```
            case 'g' :    System.out.println("Green");
                        break;
```

```
            case 'b' :    System.out.println("Blue");
                        break;
```

```
            default :    System.out.println("Weird");
```

```
        }
```

```
    }
```

```
}
```

Output: Enter character(r, g, b): g
Green

Do-While Loop

do-while: Executes a block at least once and continue iteration until condition is false.

Syntax	Flow Chart
<pre>do { statements; } while(condition);</pre>	<pre> graph TD Start([Start]) --> DoTask[Do Task] DoTask --> Condition{Condition?} Condition -- True --> DoTask Condition -- False --> End([End]) </pre>

Check Even numbers until user exits:

```
import java.util.Scanner;
class Code
{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        char res='n';
        do{
            System.out.print("Enter num : ");
            int n = sc.nextInt();
            if(n%2==0)
                System.out.println(n + " is even");
            else
                System.out.println(n + " is not even");

            System.out.print("Do you want to check another num (y/n) : ");
            res = sc.next().charAt(0);
        }while (res == 'y');
    }
}
```

Output:

```

Enter num : 5
5 is not even
Do you want to check another num (y/n) : y
Enter num : 6
6 is even
Do you want to check another num (y/n) : n
```

Introduction to OOPS

Application:

- Programming Languages and Technologies are used to develop applications.
- Application is a collection of Programs.
- We need to design and understand a single program before developing an application.

Program Elements: Program is a set of instructions. Every Program consists,

1. Identity
2. Variables
3. Methods

1. Identity:

- Identity of a program is unique.
- Programs, Classes, Variables and Methods having identities
- Identities are used to access these members.

2. Variable:

- Variable is an identity given to memory location.
or
- Named Memory Location
- Variables are used to store information of program(class/object)

Syntax	Examples
datatype identity = value;	String name = "amar"; int age = 23; double salary = 35000.00; boolean married = false;

3. Method:

- Method is a block of instructions with an identity
- Method performs operations on data(variables)
- Method takes input data, perform operations on data and returns results.

Syntax	Example
returntype identity(arguments) { body; }	int add(int a, int b) { int c = a+b; return c; }

Introduction to Object oriented programming:

- Java is Object Oriented Programming language.
- OOPs is the concept of defining objects and establish communication between them.
- The Main principles of Object-Oriented Programming are,
 1. Encapsulation
 2. Inheritance
 3. Abstraction
 4. Polymorphism

Note: We implement Object-Oriented functionality using Classes and Objects

Class: Class contains variables and methods. Java application is a collection of classes

Syntax	Example
<pre>class ClassName { Variables ; & Methods ; }</pre>	<pre>class Account{ long num; String name; double balance; void withdraw(){ logic; } void deposit(){ logic; } }</pre>

Object: Object is an instance of class. Instance (non static) variables of class get memory inside the Object.

Syntax:

ClassName reference = new ClassName();

Example:

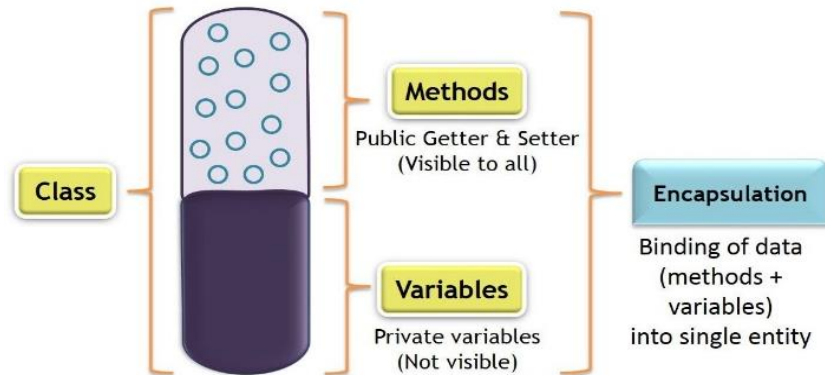
Account acc = new Account();

Note: Class is a Model from which we can define multiple objects of same type



Encapsulation:

- The concept of protecting the data with in the class itself.
- **Implementation rules:** (POJO rules)
 - Class is Public (to make visible to other classes).
 - Variables are Private (other objects cannot access the data directly).
 - Methods are public (to send and receive the data).



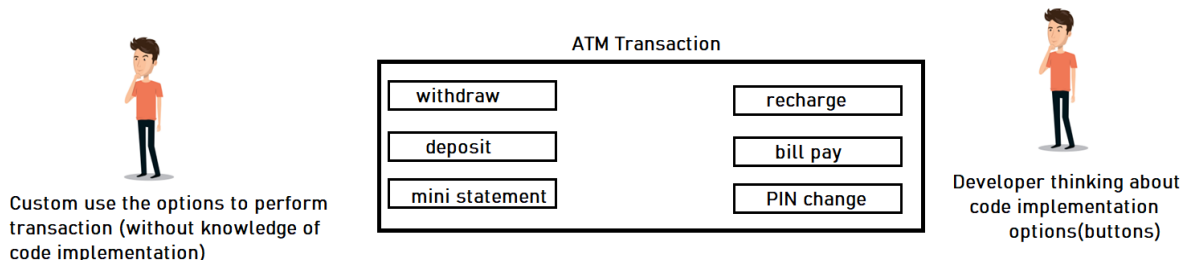
Inheritance:

- Defining a new class by re-using the members of other class.
- We can implement inheritance using "extends" keyword.
- **Terminology:**
 - **Parent/Super class:** The class from which members are re-used.
 - **Child/Sub class:** The class which is using the members



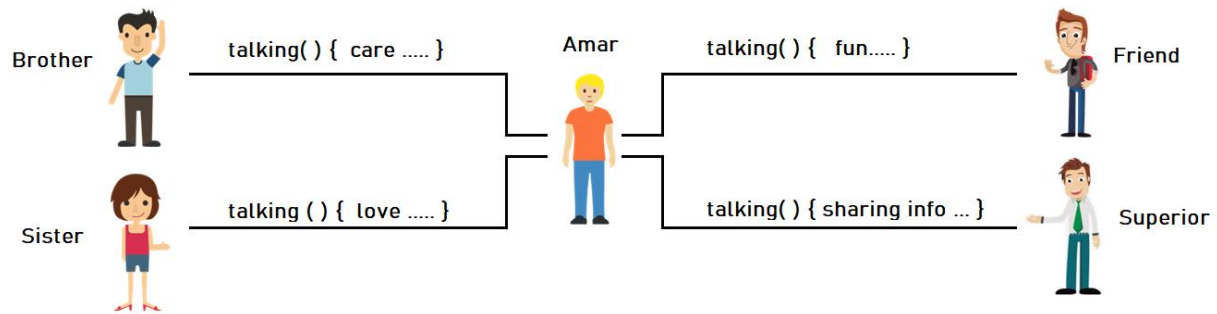
Abstraction:

- Abstraction is a concept of hiding implementations and shows functionality.
- Abstraction describes "What an object can do instead how it does it?".



Polymorphism:

- Polymorphism is the concept where object behaves differently in different situations.



Types of Relations between Classes/Objects in Java: There are three most common relationships among classes in Java that are as follows:

Use-A relation: When we create an object of a class inside a method of another class, this relationship is called dependence relationship in Java, or simply Uses-A relationship.

Has-A relation: When an object of one class is created as data member inside another class, it is called association relationship in java or simply Has-A relationship.

Is-A relation: Is-A relationship defines the relationship between two classes in which one class extends another class.

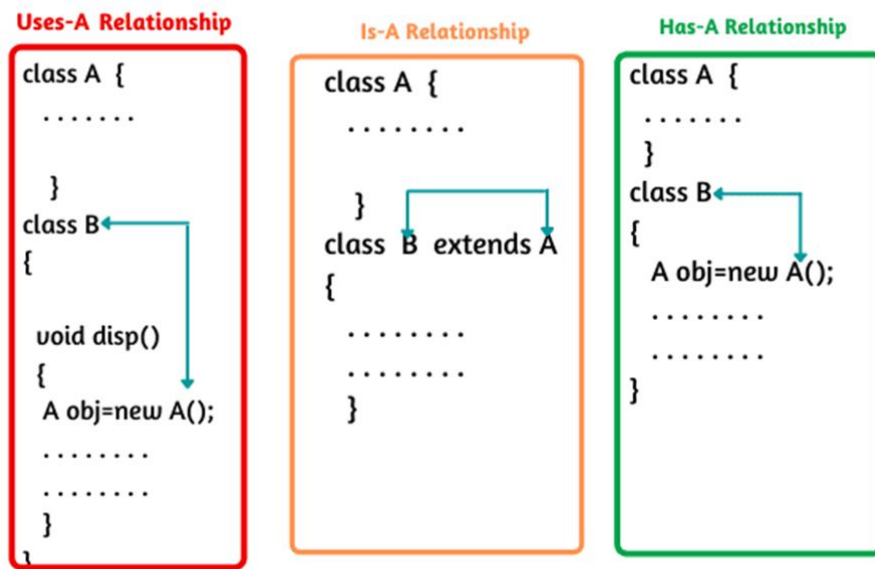


Fig: Different forms of relationship between classes in Java

Class Members

Class Members:

- The Members which we can define inside the class.
- Class Members includes
 - Variables
 - Methods
 - Blocks
 - Constructor

Variables: Variable stores information of class(object). We store information in java using 4 types of variables

1. **Static Variables:** Store common information of all Objects. Access static variables using class-name.
2. **Instance Variables:** Store specific information of Object. Access instance variables using object-reference.
3. **Method Parameters:** Takes input in a Method. Access Method parameters directly and only inside the method.
4. **Local Variables:** Store processed information inside the Method. Access local variables directly and only inside the method.

class Employee

```
{
    static String company = "Anasol";
    static String address = "Hyderabad";
    int empId;
    String empName;
    double empSalary;
    void totalSalary(double basic)
    {
        double hra = 0.2 * basic;
        double ta = 0.15 * basic ;
        double da = 0.25 * basic ;
        double total = basic + hra + ta + da ;
        System.out.println("Total Salary : " + total);
    }
}
```

-> **static variables** : store common info of all Employees

-> **instance variables** : store specific information of Employee

-> **Method parameter** : takes method input

-> **Local variables** : store processed information inside the method

Blocks:

- Block is a set of instructions without identity
- Block is either static or instance(anonymous)

Static Block: Defining a block using static-keyword. JVM invokes static block when class execution starts.

```
static
{
    statements;
}
```

Instance Block: Defining a block without static-keyword. JVM invokes instance block every time when object creates.

```
{
    statements;
}
```

Methods:

- A Block of instructions with an identity.
- Method takes input, process input and returns output
- Methods performs operations on data

Static Method: Defining a method using static keyword. We can access static methods using class-name.

```
static void display(){
    logic;
}
```

Instance Method: Defining a method without static keyword. We can access instance methods using object-reference.

```
void display(){
    logic;
}
```

Constructor: Defining a method with class name. Return type is not allowed. We must invoke the constructor in object creation process.

```
class Account{
    Account(){
        statements;
    }
}
```

Static Members in Java

Static Members:

- Defining class member with static keyword.
- Static members are:
 - Static main() method
 - Static Block
 - Static Method
 - Static Variable

Static main() method:

- Java program execution starts with main() method.
- JVM invokes the main() method when we run the program.
- We must define main() method as

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

Program to display Hello World on Console:

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

Class without main(): We can define class without main() method but we cannot run that class.

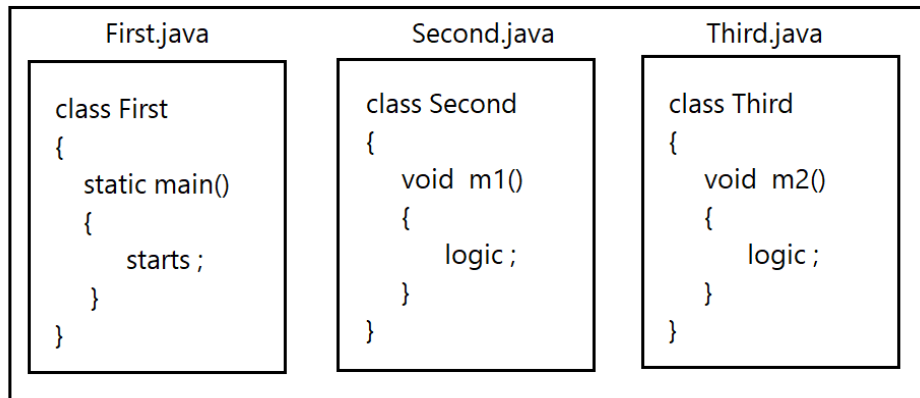
```
public class Code{
    void fun(){
        System.out.println("fun...");
    }
}
```

Runtime Error: No main() method in class Code. Please define main() method

Java Application Structure:

- We define java application in multiple java source files (.java files).
- Source file contains classes.
- Every application has only one execution point(main).
- Main() method belongs to single class from which application execution starts.

Java App

**Static block:**

- Define a block with static-keyword.
- JVM invokes when class execution starts.
- Static block executes before main() method.

```
class Pro {
    static{
        System.out.println("Static block");
    }
    public static void main(String args[]){
        System.out.println("Main method");
    }
}
```

Output:

Static Block
Main method

Methods**Method:**

- A block of instructions having identity.
- Methods takes input(parameters), process input and return output.
- Methods are used to perform operations on data

Syntax	Example
<pre>returntype identity(arguments){ statements; }</pre>	<pre>int add(int a, int b){ int c=a+b; return c; }</pre>

Classification of Methods: Based on taking input and returning output, methods are classified into 4 types.

No input – No output	with input – no output	With input – With output	No input – With output
<pre>void m1() { logic ; return ; }</pre>	<pre>void m2(int a, int b) { logic ; return ; }</pre>	<pre>char m3(String x, int y) { logic ; return 'a'; }</pre>	<pre>double m4() { logic ; return 2.34; }</pre>
<u>Invoke:</u> <pre>m1();</pre>	<u>Invoke :</u> <pre>m2(10, 20);</pre> <u>Invoke:</u> <pre>int x=10, y=20; m2(x, y);</pre>	<u>Invoke :</u> <pre>char x = m3("abcd" , 5);</pre>	<u>Invoke :</u> <pre>double d = m4() ;</pre>

Method Definition: Method definition consists logic to perform the task. It is a block.

Method Call: Method Call is used to invoke the method logic. It is a single statement.

No arguments and No return values method:

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

With arguments and No return values:

```
class Code
{
    static void main(String[] args) {
        Code.isEven(4);
        Code.isEven(13);
    }
    static void isEven(int n){
        if(n%2==0)
            S.o.p(n+" is Even");
        else
            S.o.p(n+" is Not even");
    }
}
```

With arguments with return values:

```
class Code
{
    static void main(String[] args) {
        int r1 = Code.add(10, 20);
        S.o.p(r1);
    }
    static int add(int a, int b){
        return a+b;
    }
}
```

No arguments and with return values:

```
class Code
{
    static void main(String[] args) {
        double PI = Code.getPI();
        S.o.p("PI val : " + PI);
    }
    static double getPI(){
        double PI = 3.142;
        return PI;
    }
}
```

Static Variables

Static Variable:

- Defining a variable inside the class and outside to methods.
- Static variable must define with static keyword.
- Static Variable access using Class-Name.

```
class Bank
{
    static String bankName = "AXIS";
}
```

Note: We always process the data (perform operations on variables) using methods.

Getter and Setter Methods:

- **set()** method is used to set the value to variable.
- **get()** method is used to get the value of variable.
- **Static variables** : process using static set() and get() methods
- **Instance variables** : process using instance set() and get() methods


```
class Code {
    static int a;
    static void setA(int a){
        Code.a = a;
    }
    static int getA(){
        return Code.a;
    }
    public static void main(String[] args){
        Code.setA(10);
        System.out.println("A val : " + Code.getA());
    }
}
```

Static variables automatically initialized with default values based on datatypes:

Datatype	Default Value
int	0
double	0.0
char	Blank
boolean	false
String	null

```
class Code
{
    static int a;
    static double b;
    static char c;
    static boolean d;
    static String e;
    static void values(){
        System.out.println("Default values : ");
        System.out.println("int : " + Code.a);
        System.out.println("double : " + Code.b);
        System.out.println("char : " + Code.c);
        System.out.println("boolean : " + Code.d);
        System.out.println("String : " + Code.e);
    }
    public static void main(String[] args){
        Code.values();
    }
}
```

It is recommended to define get() and set() method to each variable in the class:

```
class First{
    static int a, b, c;
    static void setA(int a){
        First.a = a;
    }
    static void setB(int b){
        First.b = b;
    }
    static void setC(int c){
        First.c = c;
    }
    static int getA(){
        return First.a;
    }
    static int getB(){
        return First.b;
    }
    static int getC(){
        return First.c;
    }
}
class Second
{
    public static void main(String[] args)
    {
        First.setA(10);
        First.setB(20);
        First.setC(30);
        System.out.println("A val : " + First.getA());
        System.out.println("B val : " + First.getB());
        System.out.println("C val : " + First.getC());
    }
}
```

Instance Members in Java

Instance Members:

- Instance members also called non-static members.
- Instance members related to Object.
- We invoke instance members using Object-address

Object Creation of a Class in java:

Syntax: `ClassName ref = new ClassName();`

Example: `Employee emp = new Employee();`

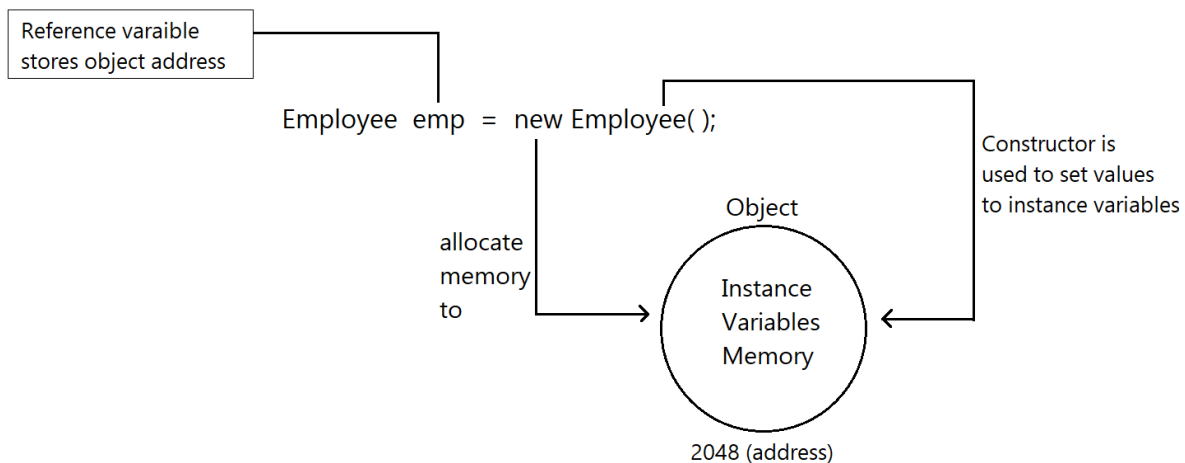
Constructor:

- Defining a method with Class-Name.
- Constructor Not allowed return-type.

```
class Employee {  
    Employee(){  
        System.out.println("Constructor");  
    }  
}
```

We must invoke the constructor in Object creation process:

```
class Employee {  
    Employee(){  
        System.out.println("Object created");  
    }  
    public static void main(String args[]){  
        Employee emp = new Employee();  
    }  
}
```



Instance Method:

- Defining a method without static keyword.
- We must invoke the method using object-reference.

No arguments and No return values method:

```
class Code
{
    public static void main(String[] args) {
        Code obj = new Code();
        obj.fun();
    }
    void fun(){
        System.out.println("fun");
    }
}
```

With arguments and No return values method:

```
class Code
{
    static void main(String[] args) {
        Code obj = new Code();
        obj.isEven(4);
        obj.isEven(13);
    }
    static void isEven(int n){
        if(n%2==0)
            S.o.p(n + " is Even");
        else
            S.o.p(n + " is Not even");
    }
}
```

With arguments and with return values method:

```
class Code {
    static void main(String[] args) {
        Code obj = new Code();
        int r1 = obj.add(10, 20);
        S.o.p(r1);
    }
    int add(int a, int b){
        return a+b;
    }
}
```

No arguments and with return values:

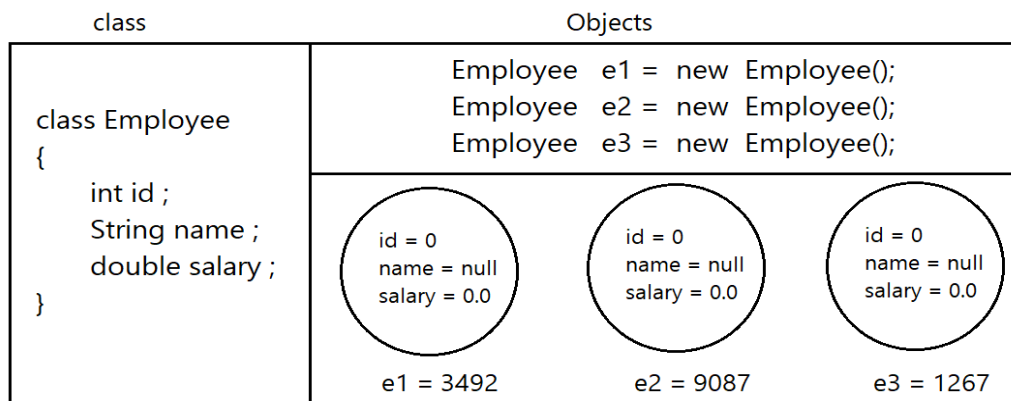
```

class Code {
    static void main(String[] args) {
        Code obj = new Code();
        double PI = obj.getPI();
        S.o.p("PI val : " + PI);
    }
    double getPI(){
        double PI = 3.142;
        return PI;
    }
}

```

Instance Variables:

- Defining a variable inside the class and outside to methods.
- Instance variables get memory inside every object and initializes with default values.



Memory allocation to objects

this:

- It is a keyword and pre-defined instance variable in java.
- It is also called **Default Object Reference Variable**.
- "this-variable" holds object address.
 - this = object_address;
- It is used to access object inside the instance methods and constructor.

Parameterized constructor:

- Constructor with parameters is called Parametrized constructor.
- We invoke the constructor in every object creation process.
- Parameterized constructor is used to set initial values to instance variables in Object creation process.

Note: We invoke the constructor with parameter values in object creation as follows
class Test

```
{
    int a;
    Test(int a){
        this.a = a;
    }
    public static void main(String[] args) {
        Test obj = new Test(10); // pass value while invoking constructor
    }
}
```

Program to create two Employee objects with initial values:

```
class Employee {
    int id;
    String name;
    double salary;
    Employee(int id, String name, double salary){
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
    void details(){
        System.out.println("Emp ID is : " + this.id);
        System.out.println("Emp Name is : " + this.name);
        System.out.println("Emp Salary is : " + this.salary);
    }
    public static void main(String[] args) {
        Employee e1 = new Employee(101, "Amar", 5000);
        Employee e2 = new Employee(102, "Annie", 8000);
        e1.details();
        e2.details();
    }
}
```

Accessing different types of variables in java:

Static Variables: Access using Class-Name.

Instance Variables: Access using Object-Name.

Local Variables: Direct access & only with in the method.

Method Parameters: Direct access & only with in the method.

Access Modifiers

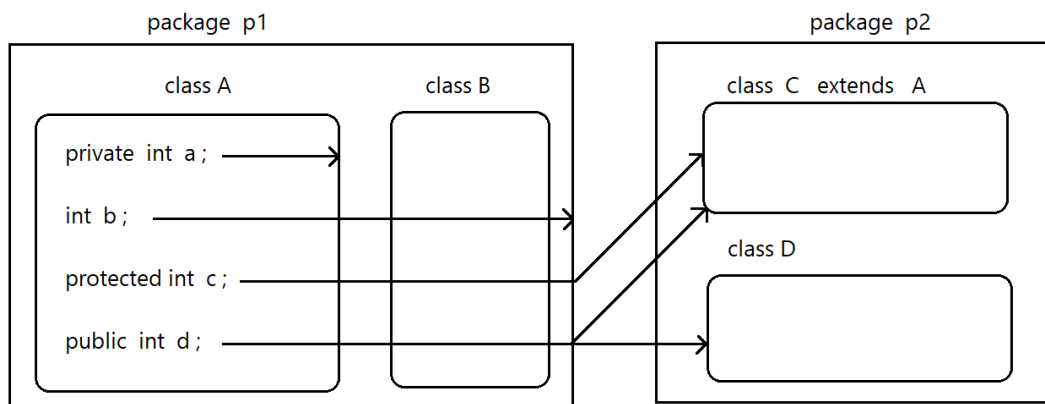
Access Modifiers:

- Access modifiers are used to set permissions to access the Class and its members (variables, methods & constructors).
- Java supports 4 access modifiers
 - private
 - <package> or <default>
 - protected
 - public

Note: We cannot apply access modifiers to blocks (we cannot access because no identity)

```
public class Pro{
    private int x;
    private Pro(){
    }
    protected void fun(){
    }
    static{
        // Error:
    }
}
```

Understanding the Access Modifiers with simple diagram:

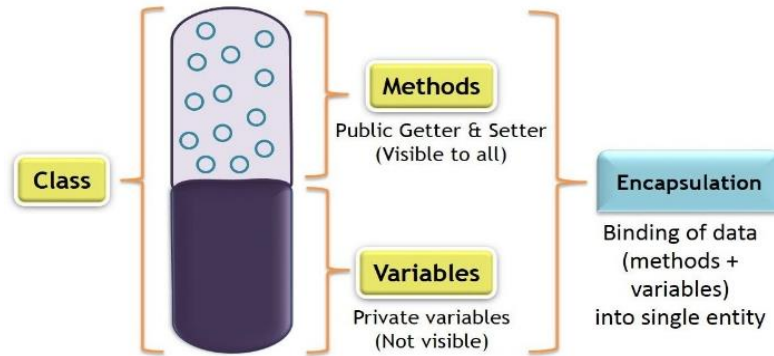


Representing the above diagram in table form:

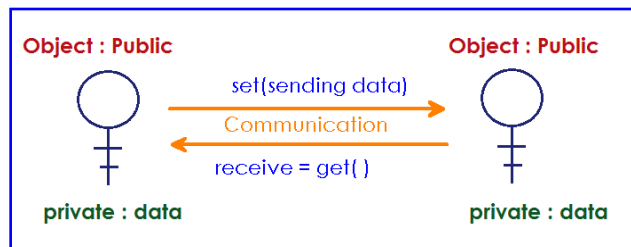
Access Modifier	Within the Class	Within the Package	Sub class of Same package	Sub class of other package	Other package
private	Yes	No	No	No	No
<default>	Yes	Yes	Yes	No	No
protected	Yes	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes	Yes

Encapsulation:

- The concept of protecting the data within the class itself.
- We implement Encapsulation through POJO rules
- POJO – Plain Old Java Object



- **Implementation rules:** (POJO rules)
 - Class is Public (to make the object visible in communication).
 - Variables are Private (other objects cannot access the data directly).
 - Methods are public (to send and receive the data).



Defining get() and set() methods to Balance variable in Account class:

Set() : takes input value and set to instance variable.

Get() : returns the value of instance variable.

```
public class Bank{
    private double balance;
    public void setBalance(double balance){
        this.balance = balance;
    }
    public double getBalance(){
        return this.balance;
    }
}
```


Employee.java: (POJO class)

```
public class Employee {  
    private int num;  
    private String name;  
    private double salary;  
    public void setNum(int num){  
        this.num = num;  
    }  
    public int getNum(){  
        return this.num;  
    }  
    public void setName(String name){  
        this.name = name;  
    }  
    public String getName(){  
        return this.name;  
    }  
    public void setSalary(double salary){  
        this.salary = salary;  
    }  
    public double getSalary(){  
        return this.salary;  
    }  
}
```

AccessEmployee.java:

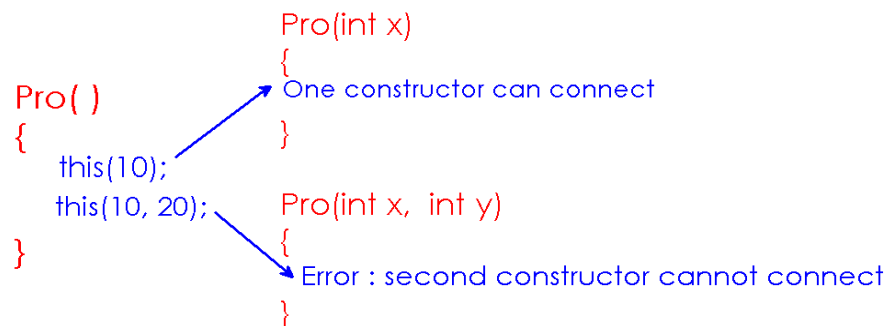
```
class AccessEmployee{  
    public static void main(String[] args) {  
        Employee e = new Employee();  
        e.setNum(101);  
        e.setName("Amar");  
        e.setSalary(35000);  
        System.out.println("Emp Num : "+e.getNum());  
        System.out.println("Emp Name : "+ e.getName());  
        System.out.println("Emp Salary : "+e.getSalary());  
    }  
}
```

Constructor Chaining:

- Invoking constructor from another constructor is called Chaining.
- this() method is used to invoke constructor.

```
class Pro
{
    Pro(){
        System.out.println("Zero args constructor");
    }
    Pro(int x){
        this();
        System.out.println("Args constructor");
    }
    public static void main(String args[]){
        new Pro(10);
    }
}
```

In constructor chaining, one constructor can be connected to only one constructor, hence call to this() must be the first statement in constructor.



Inheritance

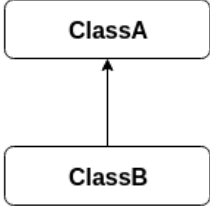
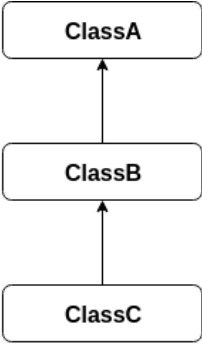
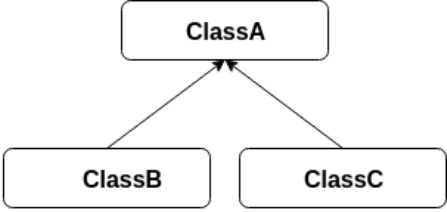
Inheritance:

- Defining a new class by re-using the members of other class.
- We can implement inheritance using "extends" keyword.
- **Terminology:**
 - **Parent/Super class:** The class from which members are re-used.
 - **Child/Sub class:** The class which is using the members

Types of Inheritance:

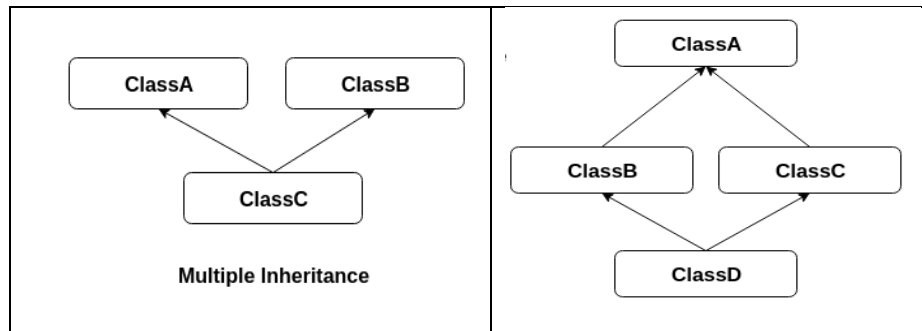
1. Single Inheritance
2. Multi-Level Inheritance
3. Hierarchical Inheritance

Note: We can achieve above relations through classes

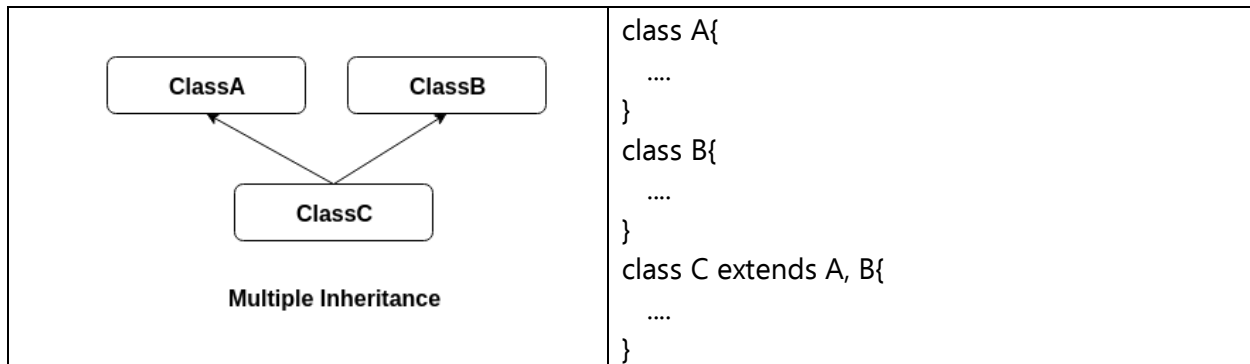
 <pre> graph BT ClassB --> ClassA </pre> <p>Single Inheritance</p>	<pre> class A{ } class B extends A{ } </pre>
 <pre> graph BT ClassC --> ClassB ClassB --> ClassA </pre> <p>Multilevel Inheritance</p>	<pre> class A{ } class B extends A{ } class C extends B{ } </pre>
 <pre> graph BT ClassB --> ClassA ClassC --> ClassA </pre> <p>Hierarchical Inheritance</p>	<pre> class A{ } class B extends A{ } class C extends A{ } </pre>

The two other inheritance types are:

1. Multiple Inheritance
2. Hybrid Inheritance

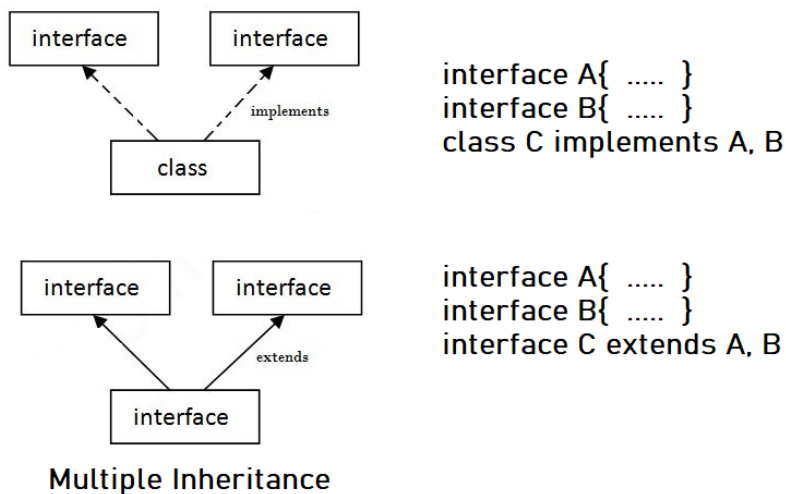


We cannot achieve multiple inheritance through Classes:

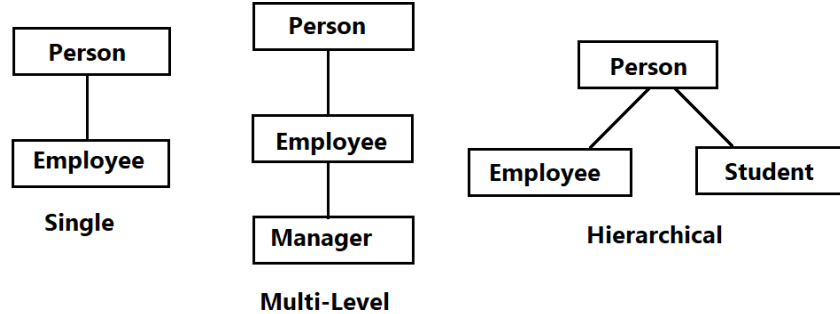


We can achieve multiple inheritance in java through interfaces:

- A class can implements more than one interface
- An interface extends more than one interface is called Multiple Inheritance



Note: We always instantiate (create object) of Child in Inheritance



Single Inheritance:

```
class Employee
{
    void doWork()
    {
        System.out.println("Employee do work");
    }
}
class Manager extends Employee
{
    void monitorWork()
    {
        System.out.println("Manage do work as well as monitor others work");
    }
}
class Company
{
    public static void main(String[] args)
    {
        Manager m = new Manager();
        m.doWork();
        m.monitorWork();
    }
}
```

In Object creation, Parent object creates first to inherit properties into Child.
We can check this creation process by defining constructors in Parent and Child.

```
class Parent{
    Parent(){
        System.out.println("Parent object created");
    }
}
```

```

class Child extends Parent
{
    Child()
    {
        System.out.println("Child object created");
    }
}
class Inheritance
{
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}

```

this	this()
A reference variable used to invoke instance members.	It is used to invoke the constructor of same class.
It must be used inside instance method or instance block or constructor.	It must be used inside the constructor.

super	super()
A reference variable used to invoke instance members of Parent class from Child class.	It is used to invoke the constructor of same class.
It must be used inside instance method or instance block or constructor of Child class.	It must be used inside Child class constructor.

super():

- In inheritance, we always create Object to Child class.
- In Child object creation process, we initialize instance variables of by invoking Parent constructor from Child constructor using super().

```

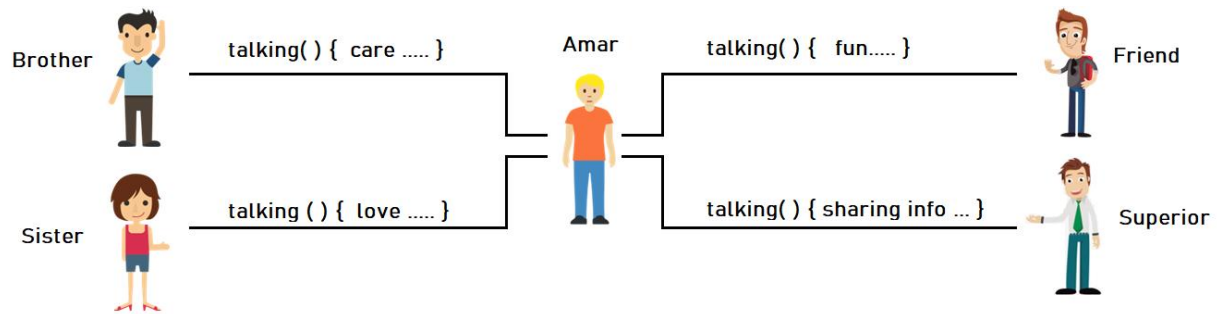
class Parent
{
    int a, b;
    Parent(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
}

```

```
class Child extends Parent
{
    int c, d;
    Child(int a, int b, int c, int d)
    {
        super(a, b);
        this.c = c;
        this.d = d;
    }
    void details()
    {
        System.out.println("a val : " + super.a);
        System.out.println("b val : " + super.b);
        System.out.println("c val : " + this.c);
        System.out.println("d val : " + this.d);
    }
}
class Main
{
    public static void main(String[] args)
    {
        Child obj = new Child(10, 20, 30, 40);
        obj.details();
    }
}
```

Polymorphism:

- Polymorphism is the concept where object behaves differently in different situations.



Polymorphism is of two types:

1. Compile time polymorphism
2. Runtime polymorphism

Compile time polymorphism:

- It is method overloading technique.
- Defining multiple methods with same name and different signature(parameters).
- Parameters can be either different length or different type.
- Overloading belongs to single class(object).

```
class Calculator
{
    void add(int x, int y) {
        int sum = x+y;
        System.out.println("Sum of 2 numbers is : " + sum);
    }
    void add(int x, int y, int z) {
        int sum = x+y+z;
        System.out.println("Sum of 3 numbers is : " + sum);
    }
}
class Main
{
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.add(10, 20);
        calc.add(10, 20, 30);
    }
}
```


println() method is pre-defined and overloaded. Hence it can print any type of data:

```
class Overload {  
    public static void main(String[] args) {  
        System.out.println(10);  
        System.out.println(12.345);  
        System.out.println('g');  
        System.out.println("java");  
    }  
}
```

Can we overload Constructor?

- Yes allowed. Overloaded constructors can connect using this() method is called Constructor chaining.

Can we overload main() method ?

- Yes. JVM invokes only standard main() method. Other main() methods must be called manually like other methods in application.

```
class Pro {  
    public static void main(String[] args) {  
        System.out.println("Standard main invoked by JVM");  
        Pro obj = new Pro();  
        obj.main();  
        Pro.main(10);  
    }  
    void main(){  
        System.out.println("No args main");  
    }  
    static void main(int x){  
        System.out.println("One arg main");  
    }  
}
```

Can we write main() as 'static public' instead of 'public static'?

- We can specify modifiers and access modifiers in any order. All the following declarations are valid in java.

public static final int a;	static public final int a;	final static public int a;
final public static int a;	static final public int a;	public final static int a;

Runtime polymorphism:

- Runtime Polymorphism is a **Method overriding technique**.
- Defining a method in the Child class with the **same name and same signature** of its Parent class.
- We can implement Method overriding only in Parent-Child (Is-A) relation.

Child object shows the functionality(behavior) of Parent and Child is called Polymorphism

```
class Parent{
    void behave(){
        System.out.println("Parent behavior");
    }
}
class Child extends Parent{
    void behave(){
        System.out.println("Child behavior");
    }
    void behavior(){
        super.behave();
        this.behave();
    }
}
class Main{
    public static void main(String[] args){
        Child child = new Child();
        child.behavior();
    }
}
```

Object Up-casting:

- We can store the address of Child class into Parent type reference variable.

Parent addr = new Child(); // upcast

or

**Child obj = new Child();
Parent addr = obj ; // upcast**

Using parent address reference variable, we can access the functionality of Child class.

```
class Parent {
    void fun(){
        System.out.println("Parent's functionality");
    }
}
```

```
class Child extends Parent{
    void fun(){
        System.out.println("Updated in Child");
    }
}
class Upcast {
    public static void main(String[] args) {
        Parent addr = new Child();
        addr.fun();
    }
}
```

Why it is calling Child functionality in the above application?

- Hostel address = new Student();
 - address.post(); -> The Post reaches student
- Owner address = new Tenant();
 - address.post(); -> The Pose reaches tenant

Down casting: The concept of collecting Child object address back to Child type reference variable from Parent type.

```
Parent p = new Parent( );
Child c = (Child)p ; ———> It is not down casting
```

```
Parent p = new Child( );
Child c = (Child)p ; ———> It is down casting
```

When we use down-casting?

- Consider a method has to return object without knowing the class name, then it returns the address as Object.
- **For example:**
 - Object getObject();
- As a programmer we collect the Object and convert into corresponding object type reference variable.
- **For example:**
 - Employee obj = (Employee)getObject();

Final Modifier in Java

final:

- Final is a keyword/modifier.
- A member become constant if we define it as final hence cannot be modified.
- We can apply final to Class or Method or Variable.

If Class is final, cannot be inherited:

```
final class A {
    // logic
}
class B extends A {    /* Error: final class-A cannot be extended */
    // logic
}
```

If Method is final, cannot be overridden:

<pre>class A{ void m1(){ ... } final void m2(){ ... } }</pre>	<pre>class B extends A{ void m1(){ ... } void m2(){ /* Error : */ ... } }</pre>
---	---

If the variable is final become constant and cannot be modified:

```
class Test {
    static final double PI = 3.142 ;
    public static void main(String[] args) {
        Test.PI = 3.1412 ; // Error : can't modify
    }
}
```

Why constructor cannot be final?

- final is used for fixing, avoid overriding and inheriting.
- Constructor cannot be final, because it can't be inherited/overridden.

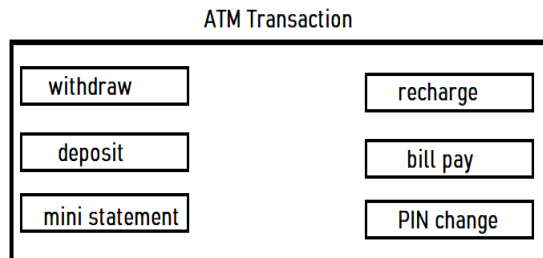
Abstraction

Abstraction:

- Abstraction is a concept of hiding implementations and shows required functionality.
- Abstraction describes "What an object can do instead how it does it?".



Custom use the options to perform transaction (without knowledge of code implementation)



Developer thinking about code implementation options(buttons)

Abstract Class:

- Define a class with abstract keyword.
- Abstract class consists concrete methods and abstract methods.

Concrete Method: A Method with body

Abstract Method: A Method without body

Class	Abstract Class
Class allows only Concrete methods	Abstract Class contains Concrete and Abstract methods
<pre>class A { void m1(){ logic; } void m2(){ logic; } }</pre>	<pre>abstract class A { void m1(){ logic; } abstract void m2(); }</pre>

Note: We cannot instantiate (create object) to abstract class because it has undefined methods.

```
abstract class Demo{
    void m1(){
        // logic
    }
    abstract void m2();
}
```

```
class Main
{
    public static void main(String[] args){
        Demo obj = new Demo(); // Error:
    }
}
```

Extending Abstract class:

- Every abstract class need extension(child).
- Child override the abstract methods of Abstract class.
- Through Child object, we can access the functionality of Parent (Abstract).

```
abstract class Parent
{
    void m1(){
        System.out.println("Parent class concrete m1()");
    }
    abstract void m2();
}
class Child extends Parent
{
    void m2(){ // override
        System.out.println("Overridden m2() method of Abstract class");
    }
}
class Main
{
    public static void main(String args[]) {
        Child obj = new Child();
        obj.m1();
        obj.m2();
    }
}
```

Initializing abstract class instance variables:

- Abstract class can have instance variables.
- Using super(), we initialize Abstract class instance variables in Child object creation process.

```
import java.util.Scanner ;
abstract class Parent
{
    String pname;
    Parent(String pname){
        this.pname = pname ;
    }
    abstract void details();
}
class Child extends Parent
{
    String cname ;
    Child(String pname, String cname) {
        super(pname);
        this.cname = cname ;
    }
    void details(){
        System.out.println("Parent Name is: "+super.pname);
        System.out.println("Child Name is : "+this.cname);
    }
}
class Main
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter Parent Name : ");
        String pname = scan.next();
        System.out.print("Enter Child Name : ");
        String cname = scan.next();
        Child obj = new Child(pname, cname);
        obj.details();
    }
}
```

Interfaces

Interface:

- Interface allow to define only abstract methods.
- Interface methods are 'public abstract' by default.

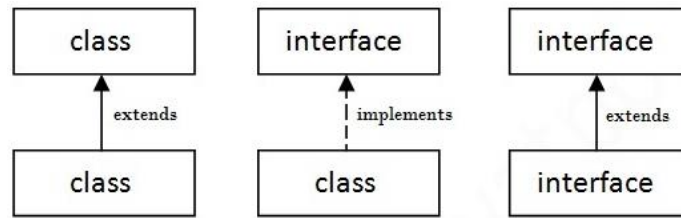
```
interface Sample {  
    void m1();  
    void m2();  
}
```

implements:

- 'implements' is a keyword.
- Interface must 'implements' by class.
- Implemented class override all abstract methods of an interface.

```
interface First {  
    void m1();  
    void m2();  
}  
  
class Second implements First{  
    public void m1(){  
        System.out.println("m1....");  
    }  
    public void m2(){  
        System.out.println("m2....");  
    }  
}  
  
class Main {  
    public static void main(String[] args){  
        First obj = new Second();  
        obj.m1();  
        obj.m2();  
    }  
}
```

Upcasting: object reference of implemented class storing into Interface type variable
--

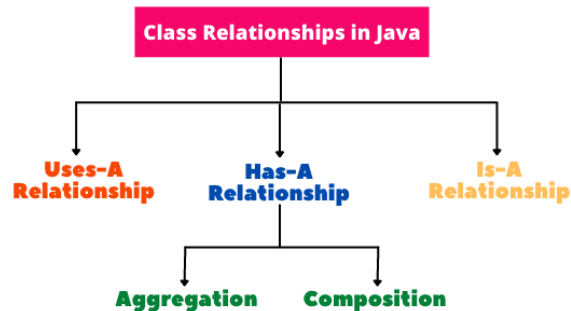
Relations:**Multiple Inheritance in java:**

- A class can extend only class
- A class can extend class and implements any number of interfaces
- An interface can extend any number of interfaces called '**Multiple Inheritance**'

	<pre> class A{ } class B{ } class C extends A, B { } Error: </pre>
	<pre> class A{ } interface B{ } class C extends A implements B { } </pre>
	<pre> interface A{ } interface B{ } interface C extends A, B { } </pre>

Objects - Relations

Three most common relationships among classes in Java:



Use-A relation: When we create an object of a class inside a method of another class, this relationship is called dependence relationship in Java, or simply Uses-A relationship.

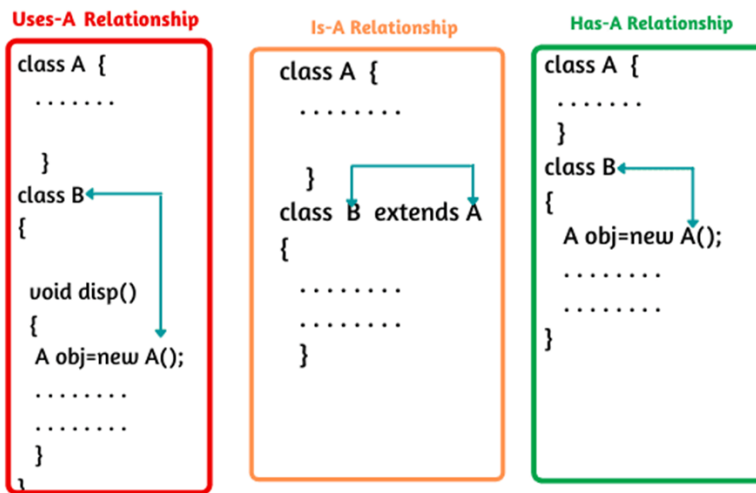
BankEmployee use Customer **Account** objects to perform their transactions

Is-A relation: Is-A relationship defines the relationship between two classes in which one class extends another class.

Every **Employee** is a **Person**
Every **Manager** is an **Employee**

Has-A relation: When an object of one class is created as data member inside another class, it is called association relationship in java or simply Has-A relationship.

Every **Person** object Has **Address** object



There are two types of Has-A relationship that are as follows:

Aggregation : Establish a weak Has-A relation between objects.
For example, **Library Has Students**.
If we destroy Library, Still student objects alive.

Composition : Establish a Strong Has-A relation between objects.
For example, **Person Has Aadhar**.
If the Person died, Aadhar will not work there after.

Association:

- Association is a relationship between two objects.
- It's denoted by "has-a" relationship.
- In this relationship all objects have their own lifecycle and there is no owner.
- In Association, both object can create and delete independently.

One to One: One Customer Has One ID
One Menu Item has One ID

One to Many: One Customer can have number of Menu orders

Many to One: One Menu Item can be ordered by any number of Customers

Many to Many: N number of customers can order N menu items

Program implementation for following Has-A relation :

```
class Address
{
    .....
}
class Person ← Has-A Relationship
{
    Address addr = new Address();
    .....
    .....
}
```

```
class Author
{
    String authorName;
    int age;
    String place;
```

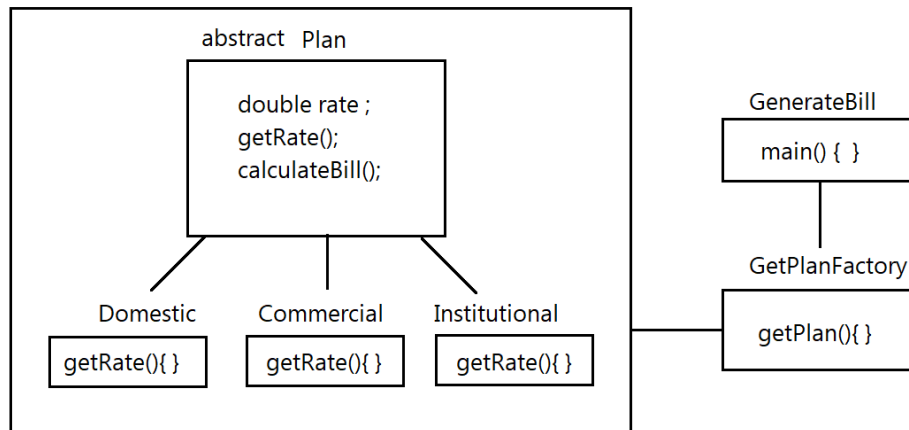
```
    Author(String name, int age, String place)
    {
        this.authorName = name;
        this.age = age;
        this.place = place;
    }
}

class Book
{
    String name;
    int price;
    Author auther;
    Book(String n, int p, Author auther)
    {
        this.name = n;
        this.price = p;
        this.auther = auther;
    }
    void bookDetails()
    {
        System.out.println("Book Name: " + this.name);
        System.out.println("Book Price: " + this.price);
    }
    void autherDetails()
    {
        System.out.println("Auther Name: " + this.auther.authorName);
        System.out.println("Auther Age: " + this.auther.age);
        System.out.println("Auther place: " + this.auther.place);
    }
    public static void main(String[] args)
    {
        Author auther = new Author("Srinivas", 33, "INDIA");
        Book obj = new Book("Java-OOPS", 200, auther);
        obj.bookDetails();
        obj.autherDetails();
    }
}
```

Factory Class and Singleton

Factory Design Pattern:

- Define interface or abstract class for creating an object but let the subclasses decide which class to instantiate.



```

import java.util.*;
abstract class Plan
{
    abstract double getRate();
    public void calculateBill(double rate, int units)
    {
        System.out.println(units*rate);
    }
}
class DomesticPlan extends Plan{
    public double getRate(){
        return 3.50;
    }
}
class CommercialPlan extends Plan{
    public double getRate(){
        return 7.50;
    }
}
class InstitutionalPlan extends Plan{
    public double getRate(){
        return 5.50;
    }
}
  
```

```
class GetPlan
{
    public Plan getPlan(String plan){
        if(plan == null){
            return null;
        }
        if(plan.equals("domestic")) {
            return new DomesticPlan();
        }
        else if(plan.equals("commercial")){
            return new CommercialPlan();
        }
        else if(plan.equals("institutional")) {
            return new InstitutionalPlan();
        }
        return null;
    }
}

class GenerateBill
{
    public static void main(String args[])
    {
        GetPlan plan = new GetPlan();
        System.out.print("Enter plan name : ");

        Scanner sc = new Scanner(System.in);
        String name = sc.next();

        System.out.print("Enter units : ");
        int units = sc.nextInt();

        Plan p = plan.getPlan(name);
        System.out.print("Bill for " + name + " plan of " + units + " units is : ");
        double rate = p.getRate();
        p.calculateBill(rate, units);
    }
}
```

Singleton Pattern: Ensure that only one instance of the class exists.

Provide global access to that instance by:

1. Declaring all constructors of the class to be private.
2. Providing a static method that returns a reference to the instance. The lazy initialization concept is used to write the static methods.
3. The instance is stored as a private static variable.

```
class Printer
{
    private static Printer printer = new Printer();
    private Printer()
    {
        // empty
    }
    public static Printer getPrinter()
    {
        return printer;
    }
    void takePrints()
    {
        System.out.println("Printing...");
    }
}
class UsePrinter
{
    public static void main(String[] args)
    {
        Printer printer = Printer.getPrinter();
        printer.takePrints();
    }
}
```

Note: We need to synchronize takePrints() method of Printer class when multiple threads trying take prints parallel.

Arrays in Java

Array:

- Array is a collection of similar data elements.
- Arrays are objects in Java.
- Arrays are static(fixed in size).

Syntax :

```
datatype identity[ ] = new datatype[size];
```

Example :

```
int arr[ ] = new int[5];
```

Declaration of array: The following code describes how to declare array variable in java. We must allocate the memory to this variable before storing data elements.

```
class Code {  
    public static void main(String[] args) {  
        int arr[];  
    }  
}
```

Allocate memory to array:

```
class Code {  
    public static void main(String[] args) {  
        int arr1[] ; // Declaration  
        arr1 = new int[5]; // memory allocation  
        int arr2[ ] = new int[5];  
    }  
}
```

length: 'length' is an instance variable of Array class. It returns the length of array.

```
class Code {  
    public static void main(String[] args) {  
        int arr[ ] = new int[5];  
        int len = arr.length ;  
        System.out.println("Length is : " + len);  
    }  
}
```

Direct Initialization of Array:

- We can initialize the array directly using assignment operator.
- We process elements using index.
- Index starts with 0 to length-1.


```
class Code {  
    public static void main(String[] args) {  
        int arr[ ] = {10,20,30,40,50};  
        System.out.println("Array elements are : ");  
        for (int i=0 ; i<arr.length ; i++){  
            System.out.println(arr[i]);  
        }  
    }  
}
```

Array stores only homogenous elements. Violation result compilation Error

```
class Code {  
    public static void main(String[] args) {  
        int arr[ ] = new int[5];  
        arr[0] = 10 ;  
        arr[1] = 20 ;  
        arr[2] = 2.34 ; // Error :  
    }  
}
```

ArrayIndexOutOfBoundsException: Accessing the location which is not in the index between 0 to Length-1 results Exception

```
class Code {  
    public static void main(String[] args) {  
        int arr[ ] = new int[5];  
        arr[0] = 10 ;  
        arr[1] = 20 ;  
        arr[6] = 30 ; // Exception : Array memory allocate at runtime.  
    }  
}
```

Read the size and construct Array Object:

```
import java.util.Scanner ;  
class Code {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        System.out.print("Enter array size : ");  
        int n = scan.nextInt();  
        int arr[ ] = new int[n];  
        System.out.println("Length is : " + arr.length);  
    }  
}
```

Program read the Size and elements of array – Print their Sum:

```
import java.util.Scanner ;
class Code {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter array size : ");
        int n = scan.nextInt();
        int arr[ ] = new int[n];

        System.out.println("Enter " + n + " values : ");
        for (int i=0 ; i<n ; i++){
            arr[i] = scan.nextInt();
        }

        int sum=0;
        for (int i=0 ; i<n ; i++){
            sum = sum + arr[i];
        }
        System.out.println("Sum of elements : " + sum);
    }
}
```

for-each loop (enhanced for loop) :

- JDK5 feature by which we can iterate Arrays and Collections easily.
- No need to specify the bounds(indexes) in for-each loop.
- **Limitation:** It can process element by element and only in forward direction.

Syntax :

```
for(<data_type> <var> : <array>or<collection>){
    Processing logic...
}
```

Program to display element using for-each loop

```
class Code {
    public static void main(String[] args) {
        int arr[ ] = {10,20,30,40,50};
        System.out.println("Array elements are : ");
        for(int ele : arr){
            System.out.println(ele);
        }
    }
}
```

Arrays Class

Arrays class:

- Java library class belongs to java.util package.
- Arrays class providing set of method to process array elements.
- Some of the methods as follows.

static void sort(int[] a)	Sorts the specified array into ascending numerical order.
static String toString(int[] a)	Returns a string representation of the contents of the specified array.
static <T> List<T> asList(T... a)	Returns a fixed-size list backed by the specified array.
static int binarySearch(int[] a, int key)	Searches the specified array of ints for the specified value using the binary search algorithm
static int[] copyOfRange(int[] original, int from, int to)	Copies the specified range of the specified array into a new array.
static boolean equals(int[] a, int[] a2)	Returns true if the two specified arrays of ints are equal to one another.
static void fill(int[] a, int val)	Assigns the specified int value to each element of the specified array of ints.

Sort array elements and display as String:

```
import java.util.Random;
import java.util.Arrays;
class Code
{
    public static void main(String[] args)
    {
        Random rand = new Random();
        int arr[] = new int[5];
        for(int i=0 ; i<5 ; i++){
            arr[i] = rand.nextInt(100);
        }
        System.out.println("Before sort : " + Arrays.toString(arr));

        Arrays.sort(arr);
        System.out.println("After sort : " + Arrays.toString(arr));
    }
}
```

Search element using Arrays.binarySearch():

```
import java.util.Scanner ;
import java.util.Arrays ;
class Code
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter array size : ");
        int n = scan.nextInt();
        int arr[ ] = new int[n];

        System.out.println("Enter " + n + " values : ");
        for (int i=0 ; i<n ; i++){
            arr[i] = scan.nextInt();
        }

        System.out.print("Enter element to be searched : ");
        int key = scan.nextInt();

        int result = Arrays.binarySearch(arr,key);
        if (result < 0)
            System.out.println("Element is not found!");
        else
            System.out.println("Element is found at index : " + result);
    }
}
```

Strings in Java

String:

- String is a sequence of characters
- String is an object in java.
- Strings must represent with double quotes.
- java.lang.String class providing pre-define methods to manipulate Strings.

String Literals: Assigning string value directly to the variable.

```
String s1 = "Hello";  
String s2 = "World";
```

String Objects: We create objects using Constructor.

```
String s1 = new String("Hello");  
String s2 = new String("World");
```

Printing String:

```
class Code {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        String s2 = "World";  
        System.out.println("S1 : " + s1 + "\nS2 : " + s2);  
    }  
}
```

length() : Instance method of String class returns the number of characters in specified string.

```
class Code {  
    public static void main(String[] args) {  
        String s = "amar";  
        System.out.println("Length of string : " + s.length());  
    }  
}
```

Program to converting Character Array to String: We can pass array as input to String constructor for this conversion.

```
class Code {  
    public static void main(String[] args){  
        char arr[] = {'a', 'm', 'a', 'r'};  
        String str = new String(arr);  
        System.out.println("String is : " + str);  
    }  
}
```

Program to convert String to Character array: toCharArray() method of String class returns the character array with specified string characters.

```
class Code {  
    public static void main(String[] args) {  
        String str = "Hello";  
        char[] arr = str.toCharArray();  
        for(char x : arr){  
            System.out.println(x);  
        }  
    }  
}
```

charAt(): String class instance method returns the character at specified index

```
class Code {  
    public static void main(String[] args) {  
        String s = "Hello";  
        System.out.println("char @ 2 is : " + s.charAt(2));  
    }  
}
```

Program to display all characters of string one by one:

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

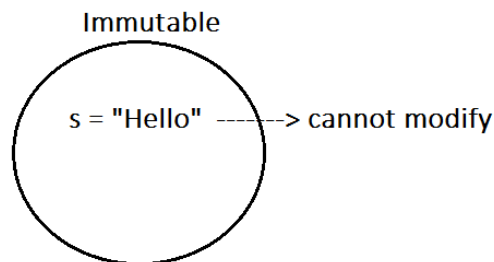
Program to count alphabets, digits and symbols in the given string:

```
class Code  
{  
    public static void main(String[] args) {  
        String str = "Online@123";  
        int alphabets=0, digits=0, symbols=0;  
        for(int i=0 ; i<str.length() ; i++){  
            char ch = str.charAt(i);  
            if( (ch>='A' && ch<='Z') || (ch>='a' && ch<='z') )  
                ++alphabets;  
            else if(ch>='0' && ch<='9')  
                ++digits;  
            else  
                ++symbols;  
        }  
    }  
}
```

```
        else
            ++symbols;
    }
    System.out.println("Alphabets : " + alphabets);
    System.out.println("Digits : " + digits);
    System.out.println("Symbols : " + symbols);
}
}
```

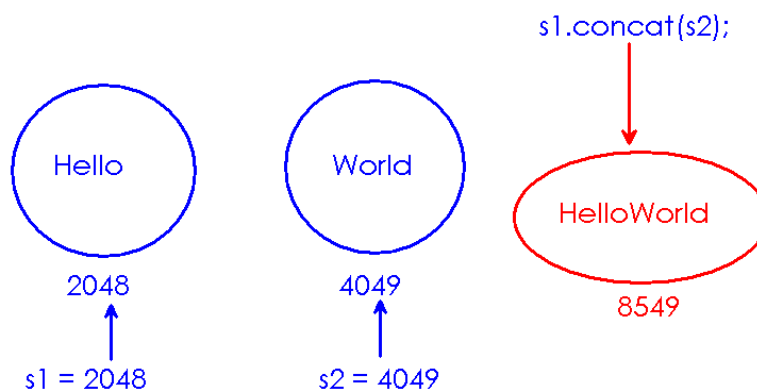
Immutable object:

- Immutable object state(data) cannot be modified.
- Immutable object state is constant.
- Strings are Immutable in java.



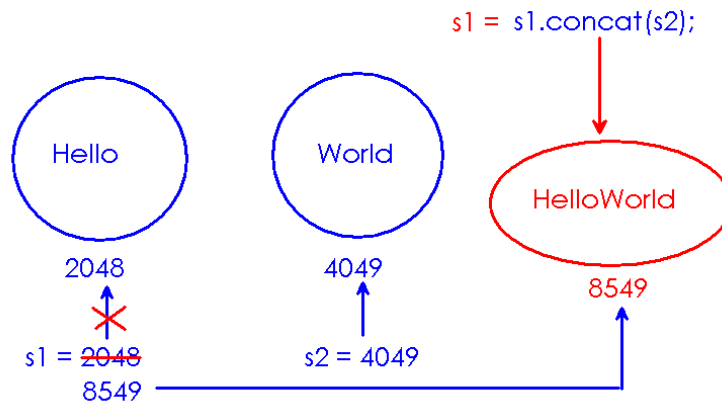
If we modify String value, new String object will be created in different memory.

```
class Code {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "World";
        s1.concat(s2);
        System.out.println("s1 : " + s1);
        System.out.println("s2 : " + s2);
    }
}
```



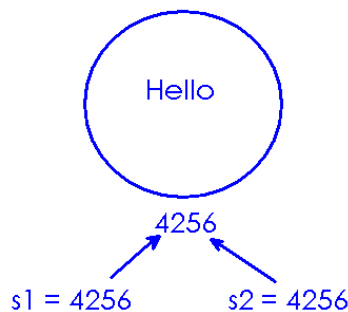
We need to collect the new String object into any variable to use:

```
class Code {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "World";
        s1 = s1.concat(s2);
        System.out.println("s1 : " + s1);
        System.out.println("s2 : " + s2);
    }
}
```



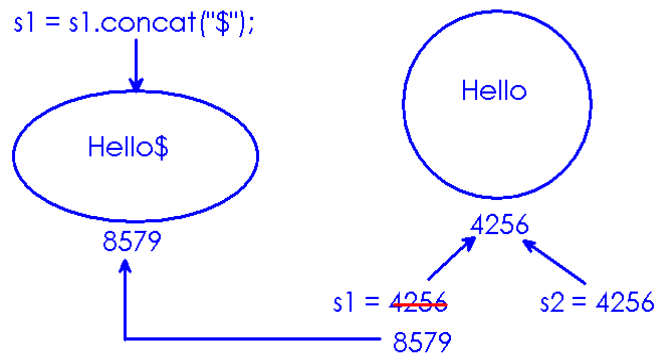
Duplicate Strings: We cannot create multiple Strings with same value(duplicates). Same object address will be shared to all String variables.

```
class Code {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "Hello";
        System.out.println("s1 location : " + s1.hashCode());
        System.out.println("s2 location : " + s2.hashCode());
    }
}
```



Question: Does s2 affects if we modify s1 in the above diagram?

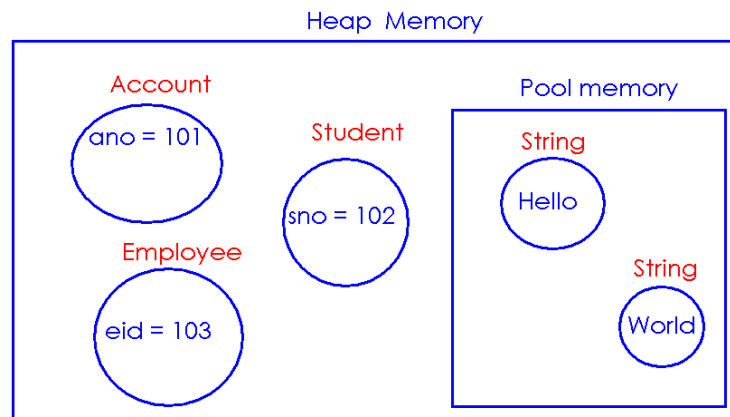
Answer: No, as strings are immutable objects, a new object will be created with modified contents.



```
class Code {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "Hello";
        System.out.println("s1 location : " + s1.hashCode());
        System.out.println("s2 location : " + s2.hashCode());
        s1 = s1.concat("$");
        System.out.println("s1 location : " + s1.hashCode());
        System.out.println("s2 location : " + s2.hashCode());
    }
}
```

String Pool Memory v/s Heap memory:

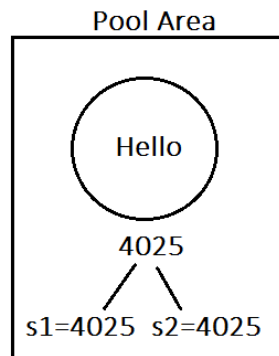
- Heap memory stores objects
- Pool memory stores only String(Immutable) objects.
- Pool memory is a part of heap memory.



String Literals:

- Always store in String Pool area.
- We can compare string literals either by using "==" operator or by using "equals()" method.

```
class Code {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        String s2 = "Hello";  
        if(s1==s2)  
            System.out.println("Equal");  
        else  
            System.out.println("Not equal");  
        if(s1.equals(s2))  
            System.out.println("Equal");  
        else  
            System.out.println("Not equal");  
    }  
}
```



String objects:

- String objects get memory in heap area.
- String object creates in heap area but the reference variable creates in Heap area.
- We cannot use == operator to check the contents of String objects.

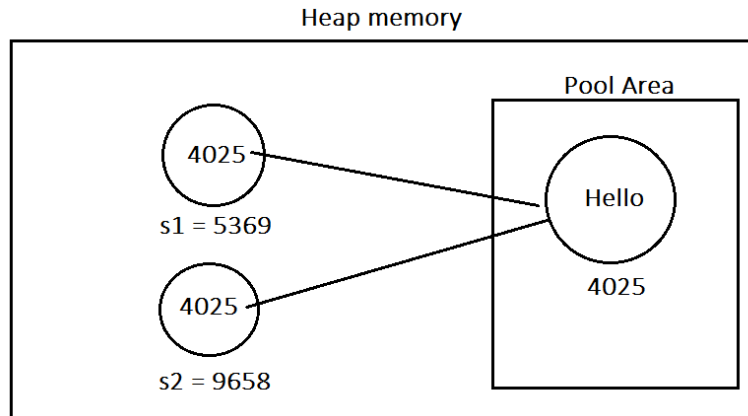
Note: It is always recommended to use equals() method to check the strings equality.

```
class Code {  
    public static void main(String[] args) {  
        String s1 = new String("Hello");  
        String s2 = new String("Hello");  
        if(s1==s2)  
            System.out.println("Equal");  
    }  
}
```

```

else
    System.out.println("Not equal");
if(s1.equals(s2))
    System.out.println("Equal");
else
    System.out.println("Not equal");
    }
}

```



split(): split() is an instance method that returns String[] array of words after split.
class Code

```

{
    public static void main(String[] args) {
        String str = "this is core java online session";
        String arr[ ] = str.split(" ");
    }
}

```

Program to count number of words in given string:

class Code

```

{
    public static void main(String[] args)
    {
        String str = "this is core java online session";
        String arr[ ] = str.split(" ");
        System.out.println("Word count : " + arr.length);
    }
}

```

Program to display words in given sentence(using for loop):

```
class Code
{
    public static void main(String[] args)
    {
        String str = "Core Java Online Session";
        String[] arr = str.split(" ");
        for(int i=0 ; i<arr.length ; i++)
        {
            System.out.println(arr[i]);
        }
    }
}
```

Program to display words in given sentence(using for each loop):

```
class Code
{
    public static void main(String[] args)
    {
        String str = "Core Java Online Session";
        String[] arr = str.split(" ");
        for(String x : arr)
        {
            System.out.println(x);
        }
    }
}
```

Program to display words in reverse order:

```
class Code
{
    public static void main(String[] args)
    {
        String str = "It is java class";
        String[] arr = str.split(" ");

        for(int i=arr.length-1 ; i>=0 ; i--)
        {
            System.out.println(arr[i]);
        }
    }
}
```

String, StringBuffer and StringBuilder

String:

- Immutable object – cannot be modified
- Thread safe by default – as we cannot modify

StringBuffer:

- Mutable object – hence we can modify the string
- Synchronized by default – hence thread safe.

StringBuilder:

- Mutable object – hence we can modify the string
- Not Synchronized by default – hence not thread safe.

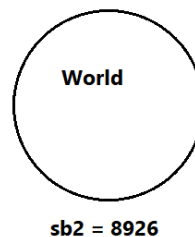
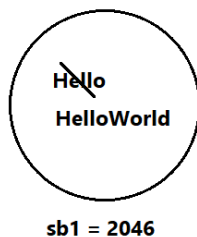
StringBuffer and StringBuilder has the same set of methods to process Strings:

class Code

```
{
    public static void main(String[] args)
    {
        StringBuffer sb1 = new StringBuffer("Hello");
        StringBuffer sb2 = new StringBuffer("World");
        sb1.append(sb2);
        System.out.println("sb1 : " + sb1);

        StringBuilder sb3 = new StringBuilder("Core");
        StringBuilder sb4 = new StringBuilder("Java");
        sb3.append(sb4);
        System.out.println("sb3 : " + sb3);
    }
}
```

sb1.append(sb2);



Exception Handling

Introduction: When executing java code, different errors can occur.

1. **Compile time errors:** Compiler raises error when we are not following language rules to develop the code.
 - Every Method should have return type.
 - Statement ends with semi-colon;
 - Invoking variable when it is not present

2. **Logical Errors:** If we get the output of code instead of Expected contains Logical error.

Expected	Result
1	12345
12	1234
123	123
1234	12
12345	1

3. **Runtime Errors:** Runtime Error is called Exception which terminates the normal flow of program execution.
 - Handling invalid input by user
 - Opening a file which is not present.
 - Connect to database with invalid user name and password.

InputMismatchException: It occurs if user enter invalid input while reading through Scanner.

```
import java.util.Scanner;
class Code {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter 2 numbers : ");
        int a = sc.nextInt();
        int b = sc.nextInt();
        int c = a+b;
        System.out.println("Sum is : " + c);
    }
}
```

Output:

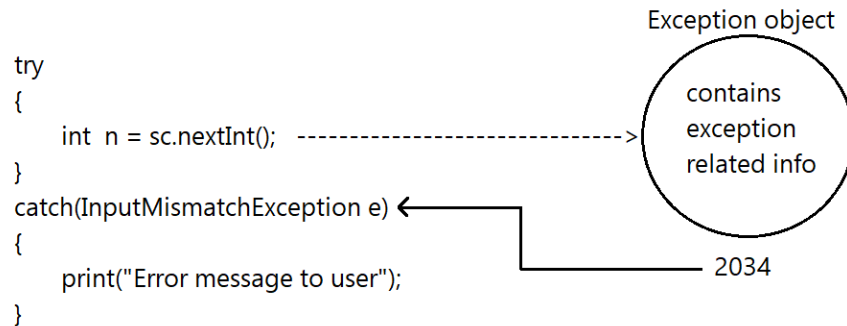
```
Enter 2 numbers :
10
abc
Exception in thread "main" : java.util.InputMismatchException
```

Handling Exception using try-catch:**try block:**

- It is used to place the code that may raise exception.
- When error occurs an exception object will be raised.

catch block:

- Exception object raised in try block can be collected in catch block to handle.

**Handling InputMismatchException:**

```

import java.util.Scanner;
import java.util.InputMismatchException;
class Code {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        try{
            System.out.println("Enter 2 numbers : ");
            int a = sc.nextInt();
            int b = sc.nextInt();
            int c = a+b;
            System.out.println("Sum is : " + c);
        }
        catch (InputMismatchException e){
            System.out.println("Exception : Invalid input given");
        }
    }
}

```

Note : Catch block executes only if exception raises in try block.

ArrayIndexOutOfBoundsException: Occurs when we try to access array elements out of index.

```

void main(){
    int[] arr = {10,20,30,40,50};
    System.out.println(arr[5]);
}

```

NumberFormatException: Occurs when we try to perform invalid data conversions.

```
void main()
{
    String s = "abc";
    int x = Integer.parseInt(s);
}
```

ArithmeticException: Occurs when try to divide any number with zero.

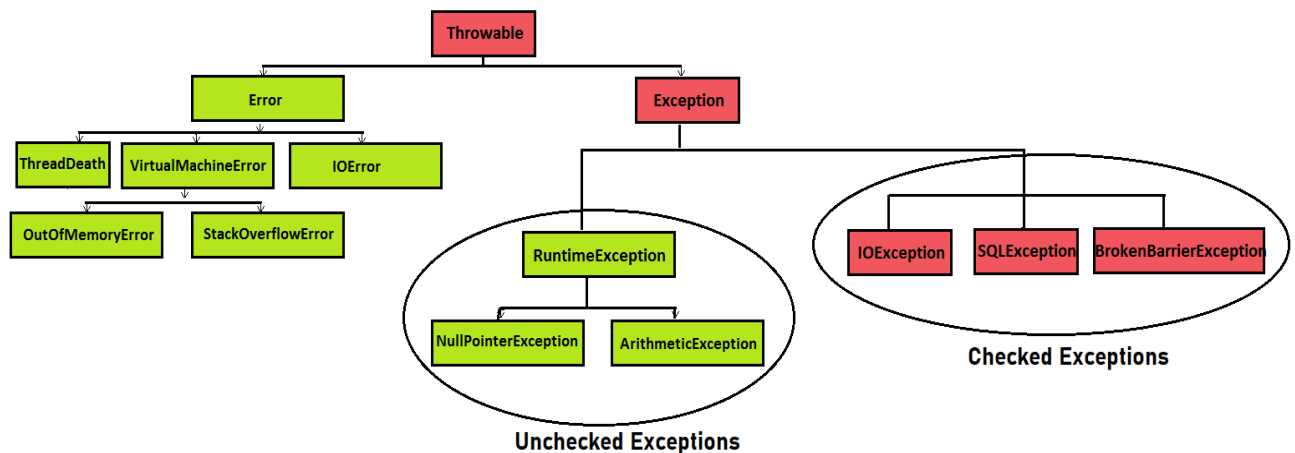
```
void main()
{
    int a=10, b=0;
    int c=a/b;
}
```

NullPointerException: Occurs when we access variable or method .

```
void main()
{
    String s = null;
    System.out.println(s.length());
}
```

Exceptions Hierarchy:

- **Throwable** is the super class of all exception classes.
- We can handle only **Exceptions** in Java not **Errors**.



Try with Multiple Catch Blocks: One try block can have multiple catch blocks to handle different types of exceptions occur in different lines of code.

Program to read 2 numbers and perform division: In this program, we need to handle two exceptions

InputMismatchException: If the input is invalid
ArithmeticException: If the denominator is zero

```
import java.util.Scanner;
import java.util.InputMismatchException;
class Code {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        try {
            System.out.println("Enter 2 numbers : ");
            int a = sc.nextInt();
            int b = sc.nextInt();
            int c = a/b;
            System.out.println("Sum is : " + c);
        }
        catch (InputMismatchException e1) {
            System.out.println("Exception : Invalid input values");
        }
        catch (ArithmeticException e2) {
            System.out.println("Exception : Denominator should not be zero");
        }
    }
}
```

We can handle Multiple exceptions in following ways:

Try with Multiple Catch blocks: It is useful to provide different message to different exceptions.	catch object using Exception class: Same logic to handle all exceptions	Using Bitwise OR: Related exceptions can handle with single catch block
<pre>try{ } catch(Exception1 e1){ } catch(Exception2 e2){ }</pre>	<pre>try { } catch(Exception e){ }</pre>	<pre>try { } catch(Exception1 Exception2 e){ }</pre>

Finally block:

- Finally, block is used to close resources (file, database etc.) after use.
- Finally block executes whether or not an exception raised.

```
class Code
{
    public static void main(String[] args)
    {
        try
        {
            // int a = 10/5 ; -> try-finally blocks execute.
            // int a = 10/0 ; -> catch-finally blocks execute.
            System.out.println("Try Block");
        }
        catch (Exception e)
        {
            System.out.println("Catch Block");
        }
        finally
        {
            System.out.println("Finally Block");
        }
    }
}
```

Why closing statements belongs to finally block?

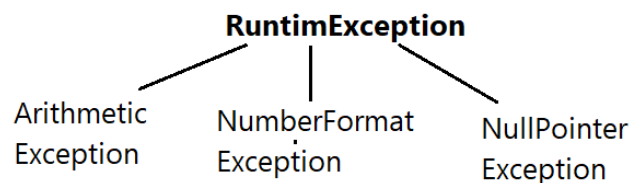
- Resource releasing logic must be executed whether or not an exception raised.
- For example, in ATM Transaction - the machine should release the ATM card in success case or failure case.

Unchecked v/s Checked Exceptions:

Unchecked Exceptions:

- The Child exceptions of RuntimeException are called Unchecked.
- Unchecked exceptions occur when program not connected to any resource.
- Handling unchecked exception is optional.
- Compiler not giving any Error if we don't handle Un checked Exceptions.

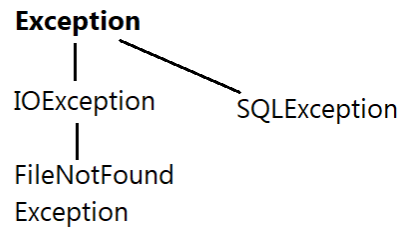
Unchecked(By Compiler)



Checked Exceptions:

- If the exception is not child of RuntimeException is called Checked.
- Checked Exceptions occur when program connected to other resource.
- Handling these exceptions is mandatory.
- If we don't handle the exception – Compiler raise Error message.

Checked(By Compiler)



throws:

- 'throws' is used to describe the exception which may raise in Method logic.
- Prototype of method specifies the Exception type.
- We handle that exception when we invoke the method.

String to int conversion:

- parseInt() is a pre-defined method that converts String to int
- parseInt() raises Exception if the input is non-convertible integer.

Pre-defined Method as follows:

```
class Integer
{
    public static int parseInt(String s) throws NumberFormatException
    {
        logic;
    }
}
```

Method-1: we can handle using try-catch while using method.

```
static void main()
{
    String s = "abc";
    try{
        int x = Integer.parseInt(s);
    }
    catch(NumberFormatException e){
        System.out.println("Invalid string");
    }
}
```

Method-2: throws the same exception without handling

static void main() throws NumberFormatException

```
{
    String s = "abc";
    int x = Integer.parseInt(s);
}
```

General Syntax to Open and Close File or Database:(programs never compiles)

- We close the resources in finally block
- We need to check the resource is connected or not while closing it.
- We use if block for this check as follows

```
if(resource != null)
{
    resource.close();
}
```

Resource – File	Resource - Database
<pre>static void main() { File f = null; try { f = new File("abc.txt"); print("File opened"); } catch (Exception e) { print("No such file"); } finally { if(f != null) { f.close(); print("File closed"); } } }</pre>	<pre>static void main() { Connection con = null; try { con = DriverManager....; print("Connected"); } catch (Exception e) { print("Connection Failed"); } finally { if(con != null) { con.close(); print("DB closed"); } } }</pre>

Custom Exceptions:

- Defining a class by extending from Exception is called Custom Exception.
- Examples:
 - InvalidAgeException
 - LowBalanceException

Defining Custom Exception:

```
class MyException extends Exception{  
    // body;  
}
```

throw:

- Pre-defined exceptions automatically raised when error occurred.
- Custom exceptions must be raised manually by the programmer using throw-keyword.

```
LowBalanceException obj = new LowBalanceException("Low Balance in Account");  
throw obj;
```

InvalidAgeException:

```
import java.util.Scanner;  
class InvalidAgeException extends Exception{  
    InvalidAgeException(String name){  
        super(name);  
    }  
}  
  
class Person{  
    static void canVote(int age) throws InvalidAgeException{  
        if(age >= 18){  
            System.out.println("Can vote");  
        }  
        else{  
            InvalidAgeException obj = new InvalidAgeException("Invalid Age");  
            throw obj;  
        }  
    }  
}  
  
class Main{  
    public static void main(String[] args){  
        System.out.print("Enter age : ");  
        Scanner sc = new Scanner(System.in);  
        int age = sc.nextInt();
```

```
        try{
            Person.canVote(age);
        }
        catch (InvalidAgeException e){
            System.out.println("Exception : " + e.getMessage());
        }
    }
}
```

LowBalanceException:

```
import java.util.Scanner;
class LowBalanceException extends Exception {
    LowBalanceException(String name) {
        super(name);
    }
}
class Account {
    int balance;
    Account(int balance) {
        this.balance = balance;
    }

    void withdraw(int amount) throws LowBalanceException {
        if(amount <= this.balance) {
            System.out.println("Collect cash : " + amount);
            this.balance = this.balance - amount;
        }
        else {
            LowBalanceException obj = new LowBalanceException("Low balance");
            throw obj;
        }
    }
}

class Bank {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter initial balance : ");
        int amount = sc.nextInt();

        Account acc = new Account(amount);
        System.out.println("Account created with balance : " + acc.balance);
    }
}
```

```
System.out.print("Enter withdraw amount : ");
amount = sc.nextInt();

try {
    acc.withdraw(amount);
}
catch (LowBalanceException e) {
    System.out.println("Exception : " + e.getMessage());
}
System.out.println("Final balance : " + acc.balance);
}
}
```

try with resources:

- The resource is as an object that must be closed after finishing the program.
- The try-with-resources statement ensures that each resource is closed at the end of the statement execution.

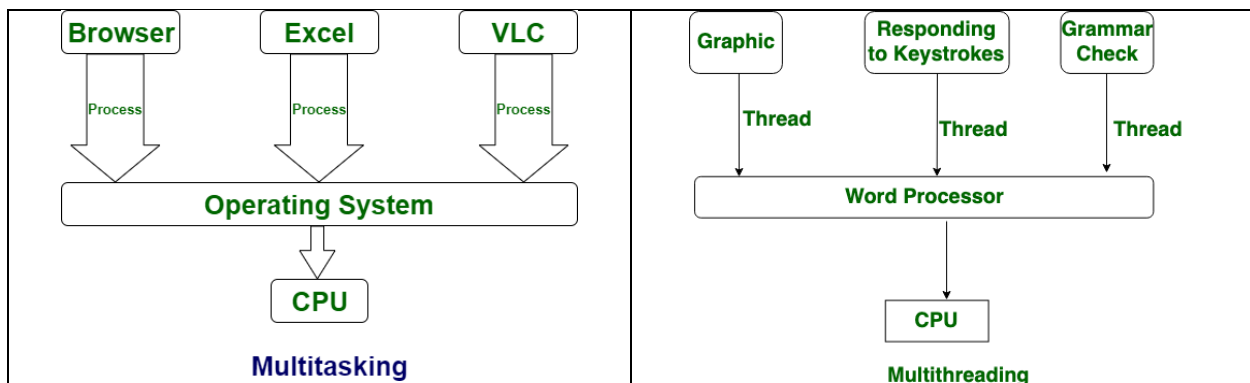
```
import java.io.*;
class Code
{
    public static void main(String[] args) throws Exception
    {
        try(FileInputStream file = new FileInputStream("Code.java"))
        {
            int ch;
            while((ch=file.read()) != -1){
                System.out.print((char)ch);
            }
        }
    }
}
```

Multi-threading

Multi-threading is a java process of executing two or more threads(programs) simultaneously to utilize the processor maximum.

Multi-tasking:

- We can implement multi-tasking in two ways.
 - Program(process) based
 - Sub program(thread) based.



Single Threaded application:

- Every Java application is single threaded by default.
- The default thread in java is "main thread"

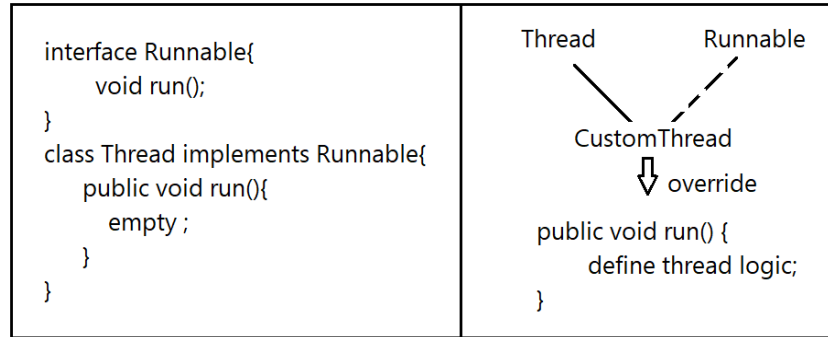
```
class Code {
    public static void main(String args[]) {
        int x=10/0;
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

Define Custom thread in java:

- We can define Custom thread in 2 ways
 - By extending Thread class
 - By implementing Runnable interface

Runnable interface has only abstract method called run().
Thread class implement Runnable interface and override run() method – but with empty body.
Programmer has to override run() method to define thread logic.



Thread Life Cycle: Every thread has undergone different states in its Life.

New State: Thread object created but not yet started.

Runnable State: Thread waits in queue until memory allocated to run.

Running State: Threads running independently once they started.

Blocked State: The thread is still alive, but not eligible to run.

Terminated State: A thread terminates either by complete process or by occurrence of error

start():

- An instance method of Thread class.
- We must start the thread by invoking start() method on Thread object.
- start() method allocates independent memory to run thread logic.

Program to define 2 Custom thread and start from main thread:

```
class Default {
    public static void main(String[] args) {
        First f = new First();
        Second s = new Second();
        f.start();
        s.start();
    }
}
class First extends Thread{
    public void run(){
        for (int i=1 ; i<=100 ; i++)
            System.out.println("First : " + i);
    }
}
class Second extends Thread{
    public void run(){
        for (int i=1 ; i<=100 ; i++)
            System.out.println("Second : " + i);
    }
}
```

sleep():

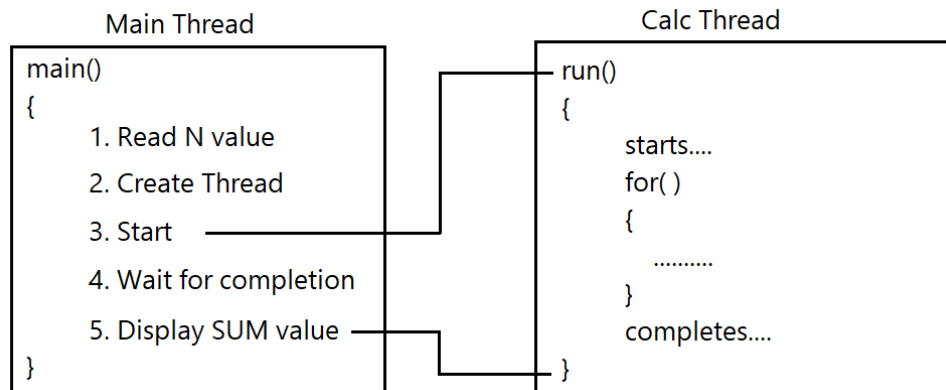
- It is a static method define in Thread class.
- It is used to stop the thread execution for specified number of milliseconds.
- sleep() method **throws InterruptedException**

```
class Default
{
    public static void main(String[] args)
    {
        First f = new First();
        Second s = new Second();
        f.start();
        s.start();
    }
}
class First extends Thread
{
    public void run()
    {
        for (int i=1 ; i<=10 ; i++)
        {
            System.out.println("First : " + i);
            try{
                Thread.sleep(1000);
            }catch(Exception e){ }
        }
    }
}
class Second extends Thread
{
    public void run()
    {
        for (int i=1 ; i<=10 ; i++)
        {
            System.out.println("Second : " + i);
            try{
                Thread.sleep(1000);
            }catch(Exception e){ }
        }
    }
}
```

join():

- An instance method belongs to Thread class.
- It stops the current the thread execution until joined thread execution completes.

Consider First thread is calculating sum of first N numbers and second thread has to display the result. The Second thread has to wait until First thread completes calculation.



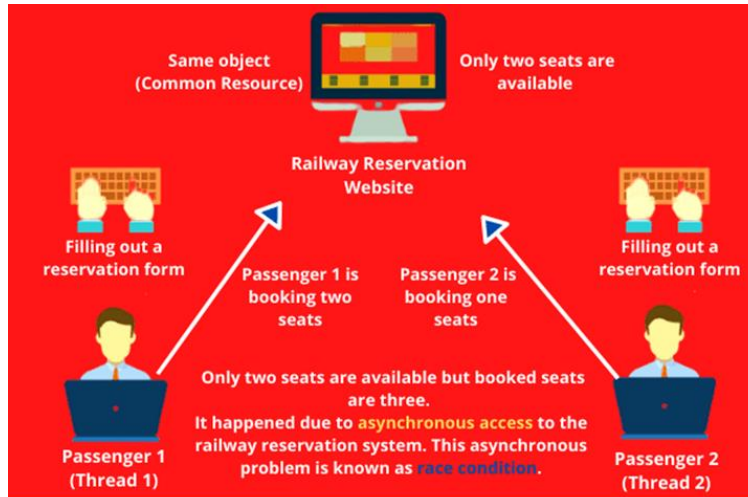
```

import java.util.Scanner;
class Main {
    static int n;
    public static void main(String[] args) {
        System.out.println("***Sum of First N numbers***");
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter n val : ");
        Main.n = sc.nextInt();
        Calc thread = new Calc();
        thread.start();
        try{
            thread.join();
        }catch(Exception e){}
        System.out.println("Sum value : " + Calc.sum);
    }
}

class Calc extends Thread {
    static int sum=0;
    public void run() {
        for (int i=1 ; i<=Main.n ; i++) {
            Calc.sum = Calc.sum+i;
        }
    }
}
  
```

Thread Synchronization:

- Synchronization is the concept of allowing thread sequentially when multiple threads trying to access the resource.
- We implement synchronization using '**synchronized**' keyword.
- We can synchronize either **Block or Method**.



Consider a method increase the value of x by 1 and when multiple threads are trying to invoke the method concurrently, we get odd results. Hence we need to synchronize the value() method.

```
class Modify {
    static int x;
    synchronized static void value() {
        x=x+1;
    }
}
class First extends Thread {
    public void run() {
        for (int i=1 ; i<=100000 ; i++) {
            Modify.value();
        }
    }
}
class Second extends Thread {
    public void run(){
        for (int i=1 ; i<=100000 ; i++) {
            Modify.value();
        }
    }
}
```

```

class Synchronization {
    public static void main(String[] args) {
        First t1 = new First();
        Second t2 = new Second();
        t1.start();
        t2.start();
        try{
            t1.join();
            t2.join();
        }catch(Exception e){}
        System.out.println("Final x value : " + Modify.x);
    }
}

```

wait(), notify() and notifyAll():

- **wait()** method forces the current thread to wait until some other thread invokes notify() or notifyAll() on the same object.
- **notify()** method for waking up threads that are waiting for an access to this object's monitor.
- **notifyAll()** method simply wakes all threads that are waiting on this object's monitor.

Syntax of Calling:

wait()	notify()	notifyAll()
<pre> synchronized(lockObject) { while(!condition) { lockObject.wait(); } //take the action here; } </pre>	<pre> synchronized(lockObject) { //establish_the_condition; lockObject.notify(); //any additional code } </pre>	<pre> synchronized(lockObject) { establish_the_condition; lockObject.notifyAll(); } </pre>

IO Streams in Java

Introduction:

- Stream is a sequence of data.
- Java I/O is used to process and input and produce results.
- The main types of streams are:

Byte Streams: are used to process 8-bit stream. We mostly use to process binary files.

Character Streams: are used to process 16-bit stream. We mostly use to process text files.

Console Streams: are used to authenticate users and passwords in Console based applications.

Buffered Streams: are used to process the data quickly.

Object Streams: are used to process objects information.

Reading from File:

- FileReader class is used to open a file in read mode by specifying the path.
- read() method is used to read character by character from file.

```
import java.io.*;
class Code {
    public static void main(String args[]) throws Exception {
        FileReader file = null;
        try {
            file = new FileReader("Code.java");
            int ch;
            while((ch=file.read())!=-1)
                System.out.print((char)ch);
        }
        finally {
            if(file !=null)
                file.close();
        }
    }
}
```

Copy File to File:

- FileWriter open specified file in Write mode.
- If the specified file is not present, it creates the file.
- write() method is used to write the information into specified file.

```
import java.io.*;
class Code{
    public static void main(String args[]) throws Exception{
        FileReader src = null;
        FileWriter dest = null;
        try{
```

```
src = new FileReader("Code.java");
dest = new FileWriter("Copy.txt");
int ch;
while((ch=src.read())!=-1){
    dest.write(ch);
}
System.out.println("Data copied");
}
finally{
    if(src !=null)
        src.close();
    if(dest != null)
        dest.close();
}
}
```

Console Streams: The Java Console class is used to get input from console. It provides methods to read texts and passwords.

The following method returns the Console object:

```
public class System{
    public static Console console();
}
```

The following methods are used to read username and password in console-based applications:

1. **String readLine()** : It is used to read a single line of text from the console.
2. **char[] readPassword()** : It is used to read password that is not being displayed on the console.

Program to read Input and Validate:

```
import java.io.*;
class Code{
    public static void main(String args[]) throws Exception{
        Console con = System.console();
        System.out.print("Enter Username : ");
        String user = con.readLine();

        System.out.print("Enter Password : ");
        char[] pwd = con.readPassword();

        String pass = String.valueOf(pwd);
        System.out.println(pass);
    }
}
```

```
        if(user.equals("Java") && pass.equals("1234"))
            System.out.println("Valid User");
        else
            System.out.println("Invalid User");
    }
}
```

BufferedStreams:

- Buffer is a temporary memory storage area.
- Buffered Streams are used to process the information quickly.
- We convert any data stream into buffer stream.

Reading a file through Buffered Streams:

```
import java.io.*;
class Code{
    public static void main(String args[]) throws Exception{
        FileReader file = null;
        BufferedReader buffer = null;
        try{
            file = new FileReader("Code.java");
            buffer = new BufferedReader(file);
            String line;
            while((line=buffer.readLine()) != null){
                System.out.println(line);
            }
        }
        finally{
            if(buffer != null)
                buffer.close();
            if(file !=null)
                file.close();
        }
    }
}
```

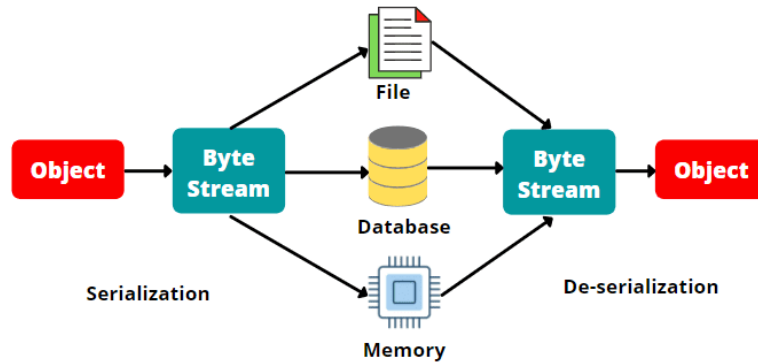
Serialization and De-Serialization in Java

Object Streams: are used to perform read and write operations of Objects.

Serialization: Writing object information into a byte-stream.

De-Serialization: Converting byte-stream into object.

To Serialize the object, the class must implements Serializable interface.
Serialized data must be stores into **.ser** file extension



transient:

- Transient is a keyword.
- Transient variables will not participate in serialization process.
- Transient variables initialized with default values after De-Serialization.

Employee.java:

```
class Employee implements java.io.Serializable{
    int id;
    String name;
    double salary;
    transient int SSN;
    Employee(int id, String name, double salary, int SSN){
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.SSN = SSN;
    }
}
```

Note: Compile the above file

Serialization.java:

```
import java.io.*;
class Serialization {
    public static void main(String[] args) throws Exception{
        FileOutputStream file = null;
        ObjectOutputStream stream = null;
        try{
            file = new FileOutputStream("result.ser");
            stream = new ObjectOutputStream(file);

            Employee emp = new Employee(101, "Amar", 35000, 1234);
            stream.writeObject(emp);
            System.out.println("Object is serialized");
        }
    }
}
```

```
    }  
    finally{  
        if(stream != null)  
            stream.close();  
        if(file != null)  
            file.close();  
    }  
}  
}
```

Note: Compile and Run the above code

De-Serialization.java:

```
import java.io.*;  
class DeSerializatio{  
    public static void main(String[] args) throws Exception{  
        FileInputStream file = null;  
        ObjectInputStream stream = null;  
        try{  
            file = new FileInputStream("result.ser");  
            stream = new ObjectInputStream(file);  
  
            Employee emp = (Employee)stream.readObject();  
            System.out.println("Object is De-serialized");  
            System.out.println("Name : " + emp.name);  
            System.out.println("SSN Number : " + emp.SSN);  
        }  
        finally{  
            if(stream != null)  
                stream.close();  
            if(file != null)  
                file.close();  
        }  
    }  
}
```

Note: Compile and Run the code

Marker Interface:

- An interface without any abstract methods. For examples Serializable & Cloneable.
- Marker interface provides information about extra behaviour of object to JVM such as
 - Object can be Serializable
 - Object can be Cloneable
- Serialization process raises Exception if the class doesn't implements Serializable interface.

Command Line Arguments

Introduction:

- Command Line is called CUI mode of OS.
- We can pass arguments (input values) to the program from command line while invoking the program.
- main() method collects all these input values and store into String type array.

main(String[] args)

Why arguments store into String type array only?

To accept any type of data values from the user.

How to read different types of values and process?

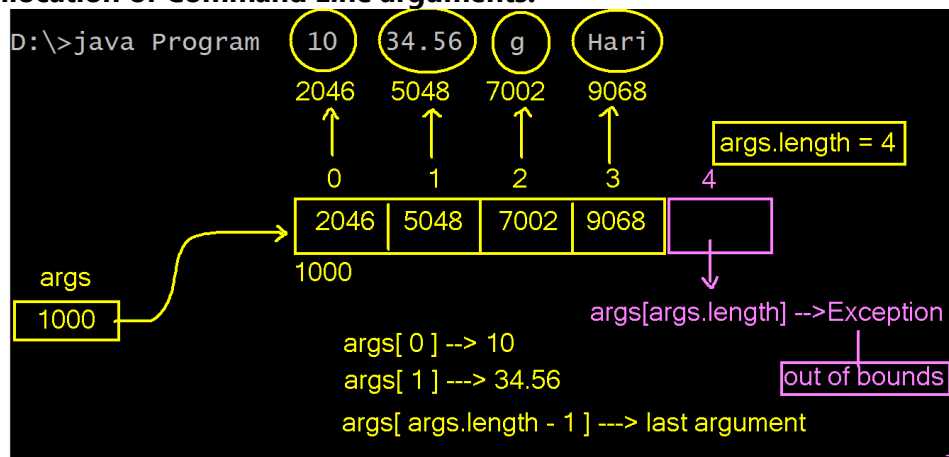
- Command line argument is by default String type.
- We convert String type data into corresponding type.
- parse() methods of wrapper classes are used to perform these data conversions.

```
public int parseInt(String s) throws NumberFormatException
public double parseDouble(String s) throws NumberFormatException
```

How to pass command line arguments?

We pass arguments followed by Program name and separated by using Space-character.

Memory allocation of Command Line arguments:



Program to print Length of Arguments:

```
class Arguments {
    public static void main(String[] args) {
        int len = list.length ;
        System.out.println("Number of arguments : "+len);
    }
}
```

Program to print all arguments:

```
class Arguments {
    public static void main(String args[] ) {
        int len = args.length ;
        if(len == 0){
            System.out.println("Please pass few arguments...");
        }
        else{
            System.out.println("List of arguments : ");
            for(int i=0 ; i<len ; i++) {
                System.out.println("args["+i+"] --> "+args[i]);
            }
        }
    }
}
```

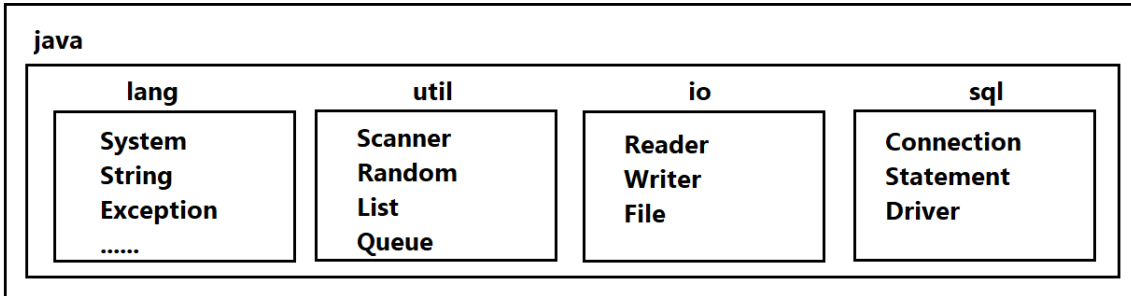
Reading values from Command Line and perform Addition operation:

```
class Code
{
    public static void main(String[] args)
    {
        if(args.length==2)
        {
            String s1 = args[0];
            String s2 = args[1];
            try
            {
                int x = Integer.parseInt(s1);
                int y = Integer.parseInt(s2);
                int z = x+y;
                System.out.println("Sum is : " + z);
            }
            catch (NumberFormatException e)
            {
                System.out.println("Error : Invalid input values");
            }
        }
        else
        {
            System.out.println("Error : Pass two numbers");
        }
    }
}
```

Packages

Package: Java library is a collection of packages. Package is a collection of related classes.

java api



import: A keyword is used to import single class or multiple classes from package into java application.

Syntax:

```
import package.*;
or
Import package.class;
```

lang package:

- lang is called default package in java.
- Without importing lang package, we can use the classes.
- Examples System, String, Exception, Thread etc.

Program to convert String to integer value:

```
class Code {
    public static void main(String[] args) {
        String s = "10";
        int x = Integer.parseInt(s);
        System.out.println("x value : " + x);
    }
}
```

Random class: Random class belongs to util package and it is used to generate Random numbers. We must import the class to use in our application.

Program to generate Random numbers from 0 to 100:

```
import java.util.*;
class Code
{
    public static void main(String[] args) {
        Random rand = new Random();
```

```
        for (int i=1 ; i<=10 ; i++){  
            int n = rand.nextInt(100);  
            System.out.println(n);  
        }  
    }  
}
```

How to create Custom package?

- We can create package by adding package statement to source code.
- Package statement must be the first statement in source file.

Code.java:

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

Syntax to compile:

```
cmd/> javac -d path filename.java
```

Example:

```
cmd/> javac -d . Code.java
```

Note the followings:

- path dot(.) represents source file location.
- -d is a command to create directory (folder).
- If the folder exists, then class file will be generated in existing folder.
- If the folder not exists, it creates the new folder and generate class file.

Run the file: We need to specify the package name to run the class

```
cmd/> java student.Code
```

Fully Qualified Name:

- Specifying the package name along with class name while using.
- We can use 'Fully Qualified Name' instead of 'import' statement

Create Object of Random class by importing:

```
import java.util.Random;
class Demo {
    public static void main(String[] args) {
        Random rand = new Random();
    }
}
```

Above code can be written without import statement:

```
class Demo {
    public static void main(String[] args) {
        java.util.Random rand = new java.util.Random();
    }
}
```

When we use Fully Qualified Name?

- The main advantage of packages is "Avoiding collisions between class names"
- We can create duplicate classes easily using package.
- To access duplicate classes from different packages, we use "Fully Qualified Name"

```
import p1.*;
import p2.*;
class Access
{
    main( )
    {
        new First(); ---> Error

        p1.First f1 = new p1.First();
        p2.First f2 = new p2.First();
    }
}
```

↓
Fully Qualified name

