Java

* High-level programming language
* Object-oriented
* Developed in 1995 by James Gosling.
* Released on 23rd January 1996.
* Used to develop console, window, web, enterprise and web applications.

Example of a java program:

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

Output:

Hello World!

Explanation:

Public - access method
Static - the method can run without creating an instance of a class
void - method doesn't return any value
String [] - method accepts a single argument i.e. string
println - prints a line
System & out - class used to access println method

Page No. :
Date :

Evolution of Java :

Version		Release Date
JDK Beta	–	1995
JDK 1.0	–	January 1996
JDK 1.1	–	February 1997
J2SE 1.2	–	December 1998
J2SE 1.3	–	May 2000
J2SE 1.4	–	February 2002
J2SE 5.0	–	September 2004
Java SE 6	–	December 2006
Java SE 7	–	July 2011
Java SE 8	–	March 2014
Java SE 9	–	September 2017
Java SE 10	–	March 2018
Java SE 11	–	September 2018
Java SE 12	–	March 2019

OOPs (Object Oriented Programming)

* Type of programming approach (method) in which everything is defined using objects that interact with one another.

* Main goal of OOPs concept in java → To better reflect real entities by representing them as objects having state (attributes / fields) and behaviour (methods).

OOPs
- Class
- Objects
- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

Class & Objects — Building blocks of OOPs

Class

* Blueprint of an object from which an object is created.
* Defines the behavior and properties of an object.
* A class can be accessed through its object.

Syntax :

A class is defined using 'class' keyword.

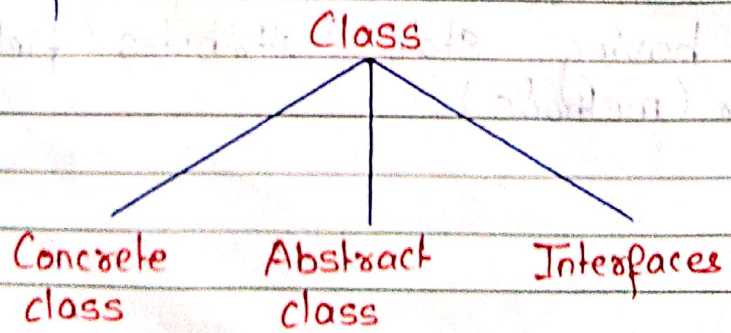
```
class class_name
```

```
class class_name {
    // class body
}
```

Types Example :

```
class Mystudent {
    ---------
}
```

Types of classes :



Page No. :
Date :

1. Concrete class :- A normal class containing methods, class variables, constructors, blocks and everything is called concrete class.

Example :

```
class Mystudent {
    // class body
}
```

2. Abstract class

* A class which have at least one abstract method
* Defined using abstract keyword

abstract method → method without body

* Abstract classes can't be instantiated but can only be inherited.

Example :

```
abstract class Mystudent {
    // abstract method
    abstract void show();
}
```

3. Interfaces

* classes which contain method signatures and fields.
* It should be implemented by a class.

Example :

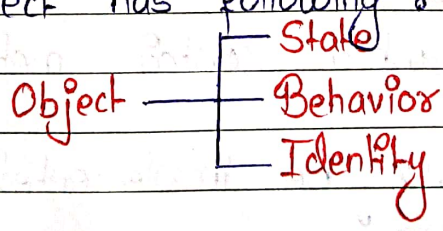
Example :

```
public interface Animal {
    public void eat();
    public void run();
}

public class Dog implements Animal {
    public void eat() {
        // implementation
    }
    public void run() {
        // implementation
    }
}
```

Object

* object is a real-world entity.
* object has its own property and behavior.
* A program can have many projects as per the requirement.

An object has following :



State : Represents attributes & properties of object

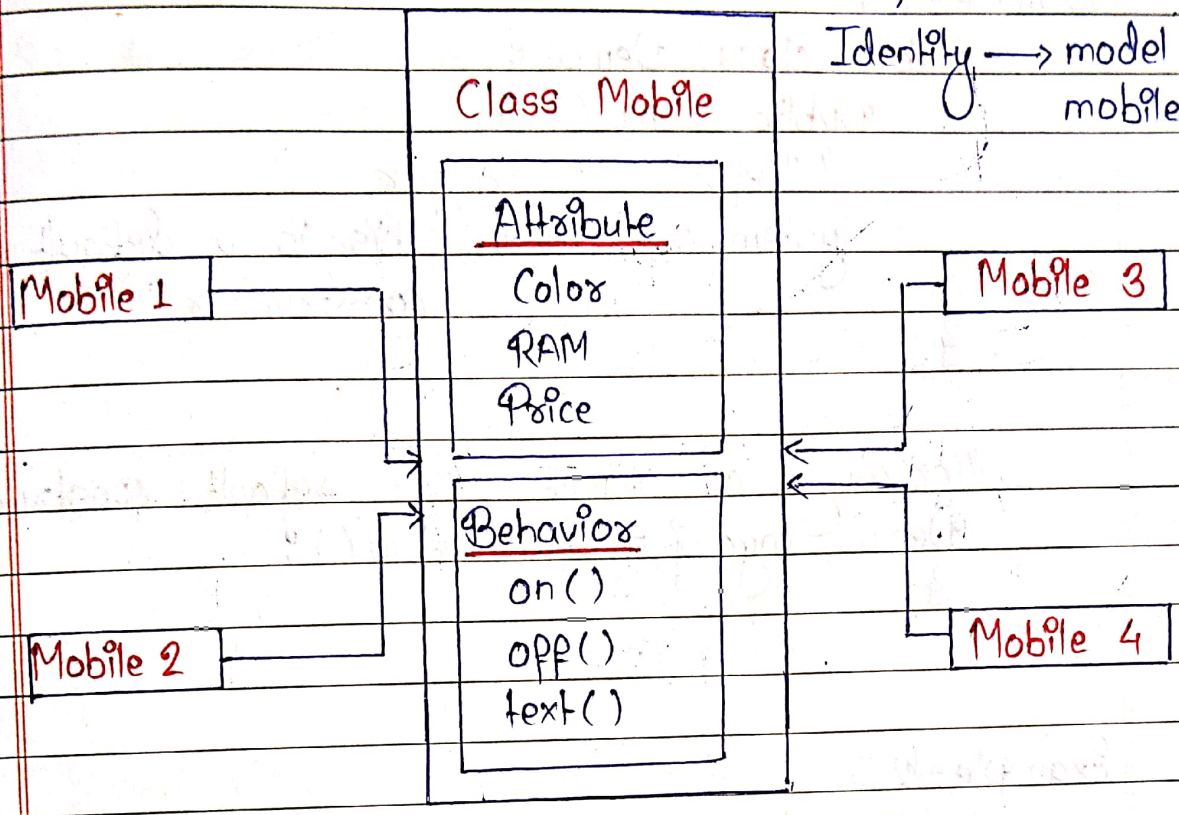
Behavior : Defines methods (function) of an object

Identity : provides a unique name to an object & also enables communication between two or more objects.

Page No. :
Date :

Here,

Identity → model of mobile



Creating object in java :

i) Declaration :- object is declared using a variable with the class name as the data type.

ii) Instantiation :- object is instantiated using the 'new' keyword.

iii) Initialization :- object is initialized using the by calling the class constructor.

~~Example - 1 :-~~

Example - 1 :

```
Public class Demo {
    Public Demo() {
        // Default constructor
        System.out.println("This is a default
                            constructor");
    }

    // creating an object using default constructor
    Demo myobj = new Demo();
    }
}
```

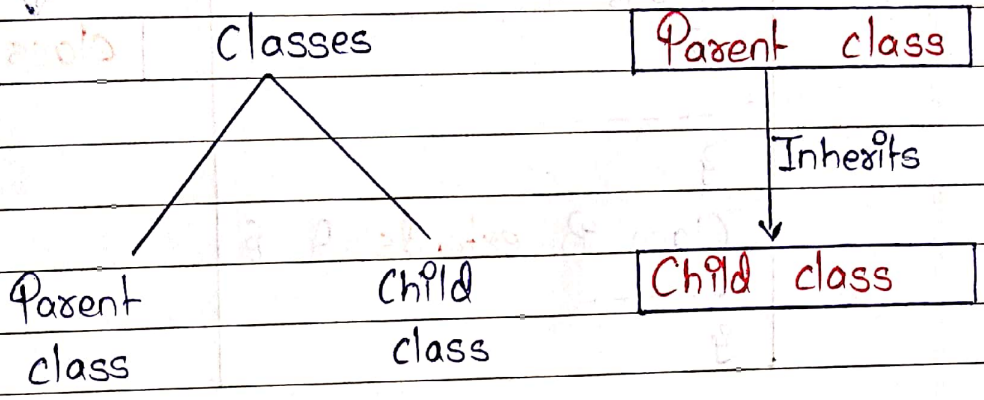
Example - 2 :

```
Public class Demo(String name) {
    // This constructor has no parameter
    System.out.println("Hello :" + name);
    System.out.println("Welcome to Java
                        Tutorials");
    }

    // creating an object using parameterized
    constructor
    Demo myobj = new Demo("World");
}
```

Page No. :
Date :

Inheritance :- Inheritance is a feature of OOPs where the properties of one class can be inherited by the other.

The are two kind of classes :

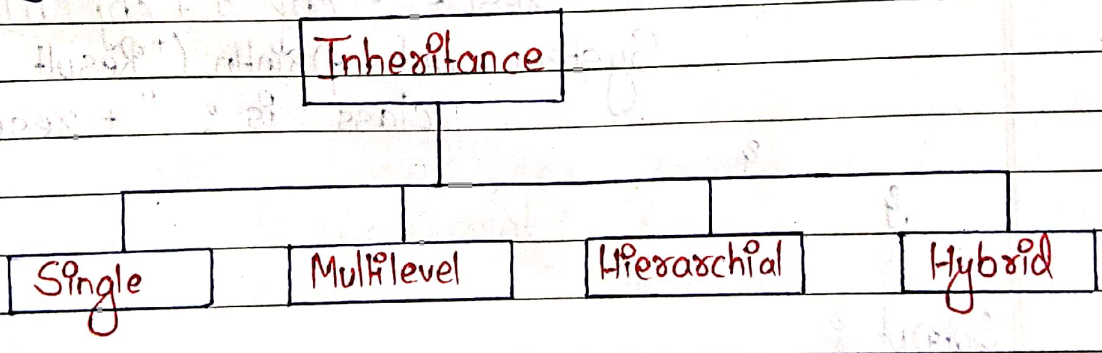


Parent class ⟶ Super class ⟶ Base class
Child class ⟶ Sub class ⟶ Derived class

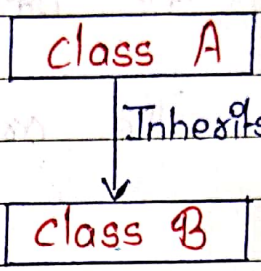
Parent class :- class through which the property is inherited.

Child class :- class through which inherits the properties of parent class.

Types of inheritance :



1. Single Inheritance :- One class inherits another class. Only single parent class and single child class.



Syntax :

```
Class A
{
-----
}
Class B extends A {
------
}
```

Example :

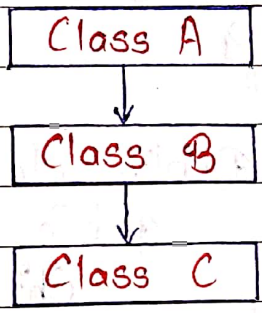
```
Public class superclass {
    int a = 10;
    int b = 5;
}

class subclass extends superclass {
    Public static void main (String [] args) {
        subclass obj = new subclass ();
        int result = (obj.a + obj.b);
        System.out.println ("Result of child
                class is : " + result);
    }
}
```

Output :

Result of child class is : 15

Page No. :
Date :

2. Multilevel Inheritance :- When a class is derived is derived from one class and another class is derived from that derived class is known as multilevel inheritance.



Syntax :

```
Class A {
---------
}
Class B extends A {
---------
}
Class C extends B {
-----
}
```

Example :

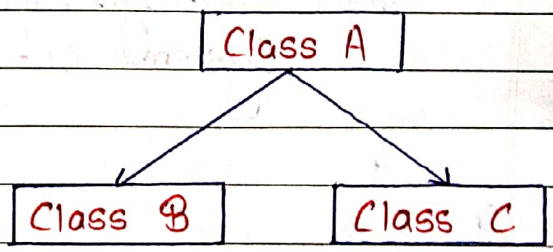
```
class Grandparent {
    void displayGrandparent() {
        System.out.println("This is the Grand-
                            parent class.");
    }
}

class parent extends Grandparent {
    void displayparent() {
        System.out.println("This is the
                            parent class.");
    }
}
```

```
class Child extends Parent {
    void displayChild() {
        System.out.println("This is the Child class.");
    }
}

Public class MultilevelInheritanceExample {
    public static void main(String[] args) {
        Child child = new Child();
        child.displayGrandParent();
        child.displayParent();
        child.displayChild();
    }
}
```

3. Hierarchial inheritance :- When a parent class has more than one (multiple) child classes, it is known as hierarchial inheritance.

Syntax :

```
Class A {
------
}
Class B extends A {
-------
}
Class C extends A {
-----
}
```



Page No. :
Date :

Example :

```
Public class Animal {
    void eat() {
        System.Out.println("Animal can eat");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

Class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows");
    }
}

class HierarchialInheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();

        dog.eat();
        dog.bark();

        cat.eat();
        cat.meow();
    }
}
```

Page No. :
Date :

Output :

Animal can eat
Dog barks
Animal can eat
Cat meows

Here,
- Animal is the parent class.
- Dog and Cat are two different child classes that inherit from Animal class. Th

4. Hybrid Inheritance :- Hybrid Inheritance is a combination of multiple inheritance and multilevel inheritance.

Hybrid Inheritance = Multiple Inheritance + Multilevel Inheritance

This type of inheritance can be achieved through the use of interfaces.

Encapsulation :- Encapsulation is a mechanism where the data and code is binded together as a single unit. It also means to hide our data in order to make it safe from any modification. The methods and variables of a class are well hidden and safe through encapsulation.

Page No. :
Date :

Encapsulation

Class ⟶ | Methods | Variables |

We can achieve encapsulation in Java by:
- Declaring the variables of a class as private
- Providing public setter and getter methods to modify and view the variable values.

Code:

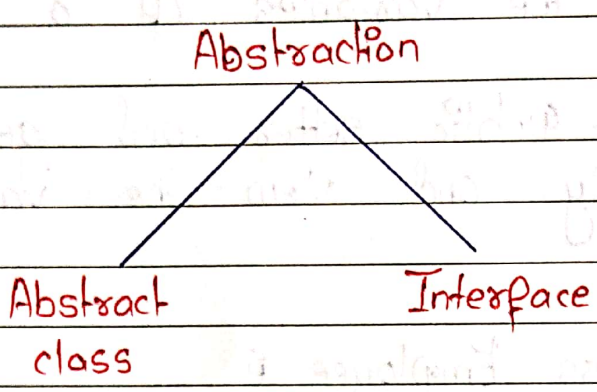
```
public class Employee {
    private String name;
    public String getName() {
    return name;
    }
    public void setName(String Name) {
     this.name = name;
    }
    public static void main(String[] args) {
    }
}
```

Explanation :- We have created a class Employee which has a private variable 'name'. We have then created a getter and setter methods through which we can get and set the name of an employee. Through these methods, any class which wishes to access the name variable has to do it using these getter and setter methods.

Page No. :
Date :

Abstraction :- Abstraction refers to the process of hiding the internal details and showing the essential things to the user. It helps to reduce complexity.

We can achieve abstraction in two ways :



1) Abstract class

- Abstract class is a class which cannot be instantiated i.e. its object cannot be created.
- 'Abstract' keyword is used to create an abstract class.
- It is used to declare common characteristics of subclasses.
- It can only be used as a superclass for other classes that extend the abstract class.
- Abstract class also contains fields and methods like any other classes.
- Abstract class's objects cannot be created but its reference can be done.
- Abstract method is used in abstract class.

Syntax :

```
Abstract class class-name {
---
}
```

Page No. :
Date :

Example - 1 :

```
abstract class Person {
    private String name;
    private int Age;
    public void setName (String n) {name = n;}
    public void setAge (int a) {age = a;}
}
class AbstractExample1 {
    public static void main (String [] args)
    {
        Person p = new Person (); // can't be instantiated
    }
}
```

Example - 2 :

```
abstract class shape {
    public abstract void draw (); // abstract method
    public void display () // concrete method
    {
        System.out.println ("This is a shape.");
    }
}

class circle extends shape {
    public void draw () {
        System.out.println ("Drawing a circle.");
    }
}
```

: glimpsed

```
class Rectangle extends Shape {
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public class AbstractExample {
    public static void main(String[] args) {
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();

        circle.display();
        circle.draw();

        rectangle.display();
        rectangle.draw();
    }
}
```

Here,

Shape → abstract class
draw() → abstract method
display() → Concrete method
Circle & Rectangle → subclasses which extends class shape
circle → object of Circle
rectangle → object of class Rectangle

```
This is a shape
Drawing a circle
This is a shape
Drawing a rectangle
```

Page No. :
Date :

2) Interface

- Java doesn't support multiple inheritance. That's why, Interface is used to achieve multiple inheritance in java.
- An interface is basically a kind of a class.
- Like class, interface also contains methods & functions but it contains only abstract methods & final fields.

Abstract method → method i.e. declared without an implementation (i.e. body)

Final fields → A final field cannot change its value

Syntax :

```
Interface Interface_name
{
// variable declaration ;
// method declaration ;
}
```

Example :

```
Interface values
{
static final int code = 101 ;
static final string name = "Kajal" ;
void display ();
}
```

Page No. :
Date :

Extending Interfaces :- As we extends a class in Inheritance. In the same way, an interface extends another interface.

Syntax :

```
Interface name1 extends name2
{
body of name1;
}
```

Example :

```
interface abc
{
Int code = 10;
String str = "Kajal";
}

interface xyz extends abc
{
void display();
}
```

Note :

~~When a class ——→ an interface ——→ 'imp~~

One class —extends→ Another class

~~A~~ Class —implements→ interface

Implementing Interfaces :- Interfaces are made for performing the work of a superclass. It is a way to implement multiple Inheritance in java.

Page No. :
Date :

Syntax :

```
Class class_name implements interface_name
{
body of the class
}
```

Syntax :

```
Class class_name extends superclass implements
                                        interface_name
{
body of the class
}
```

Page No. :
Date :

~~Polymorphism~~ Polymorphism :- Polymorphism means having various forms. 'Poly' means 'many' and 'morph' means 'forms'.

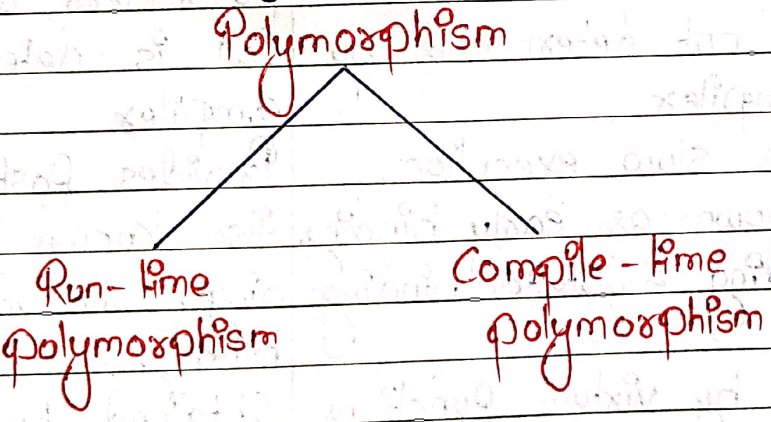
or

Polymorphism = Poly + Morph

Poly → various
Morph → forms

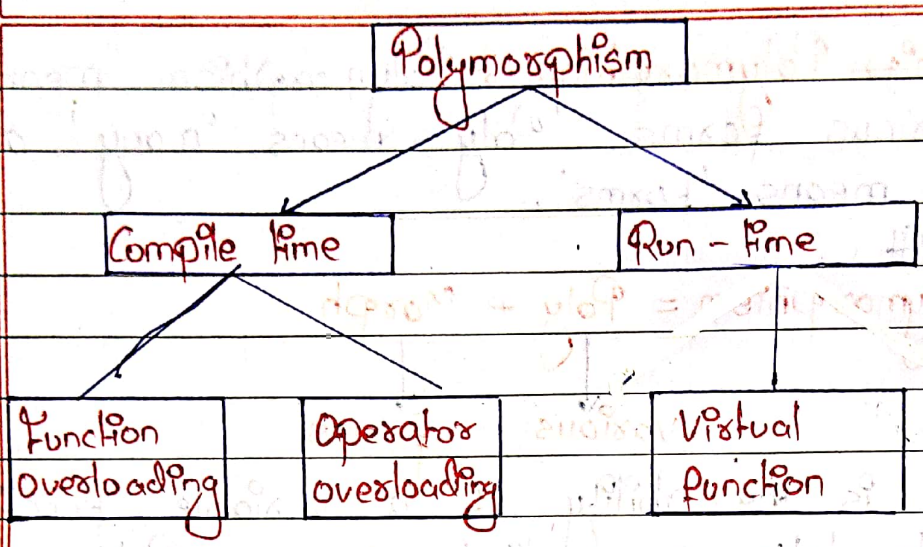
It is the ability of a variable, function or object to take on multiple forms. Polymorphism allows us to define one interface or method and have multiple implementations.

Types of Polymorphism :



Run-time polymorphism
- Run-time polymorphism is a polymorphism in which the overridden method is resolved at run-time.

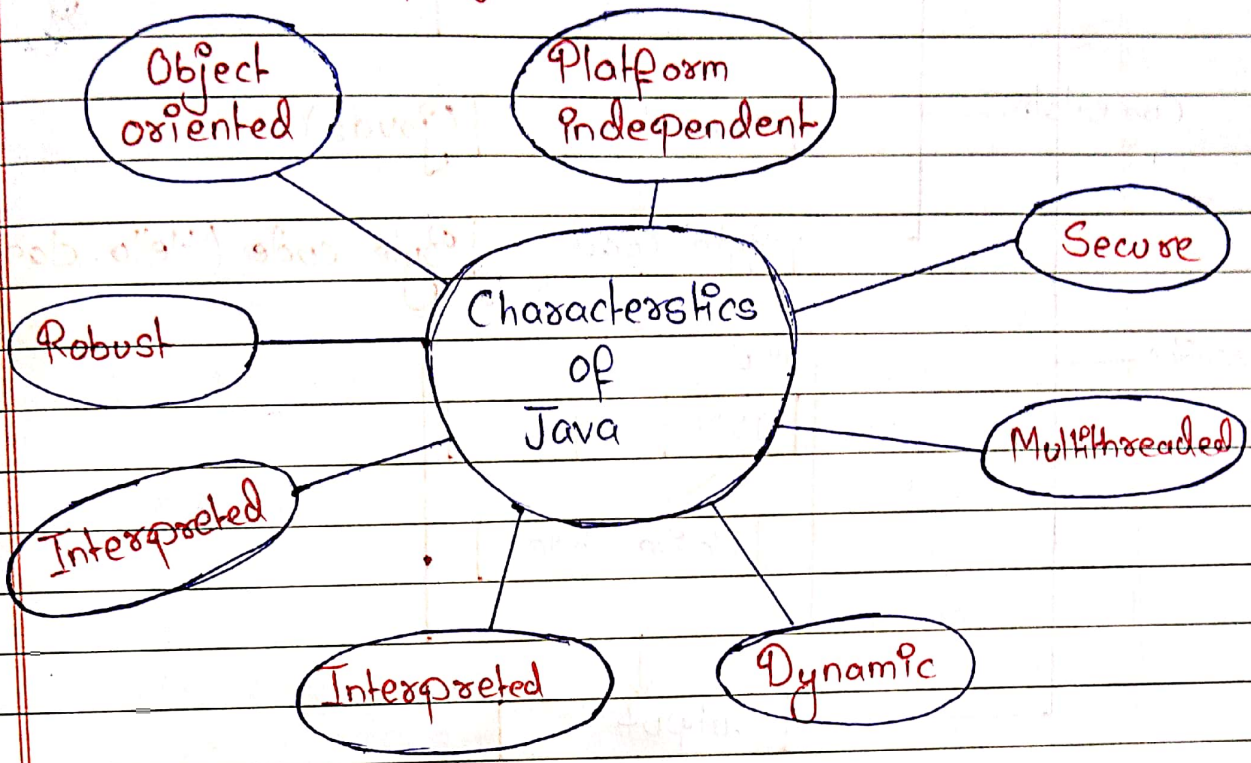
Compile time polymorphism
- Compile time polymorphism is a polymorphism in which the overloaded method is resolved at compile time.



Run-time polymorphism	Compile-time polymorphism
- It allows us to have different classes but same method & same signature name.	It allows us to have more than one method with same name but different parameters or return types.
- Call is not determined by the compiler.	Call is determined by the compiler.
- Provides slow execution.	Provides fast execution.
- Also known as late binding, overriding & dynamic binding.	Also known as early binding, overloading & static binding.
- Obtained by virtual functions.	Obtained by operator overloading & function overloading.
- No inheritance involved.	Inheritance is a factor.

Page No. :
Date :

Characterstics of java :



Compilation and Execution process of a java program :

The java compilation and execution process consists of the following five steps :

1. Creation of a java program.
2. Compiling a java program.
3. Byte code ~~gen~~ formation.
4. Verification for byte code.
5. Java program execution.

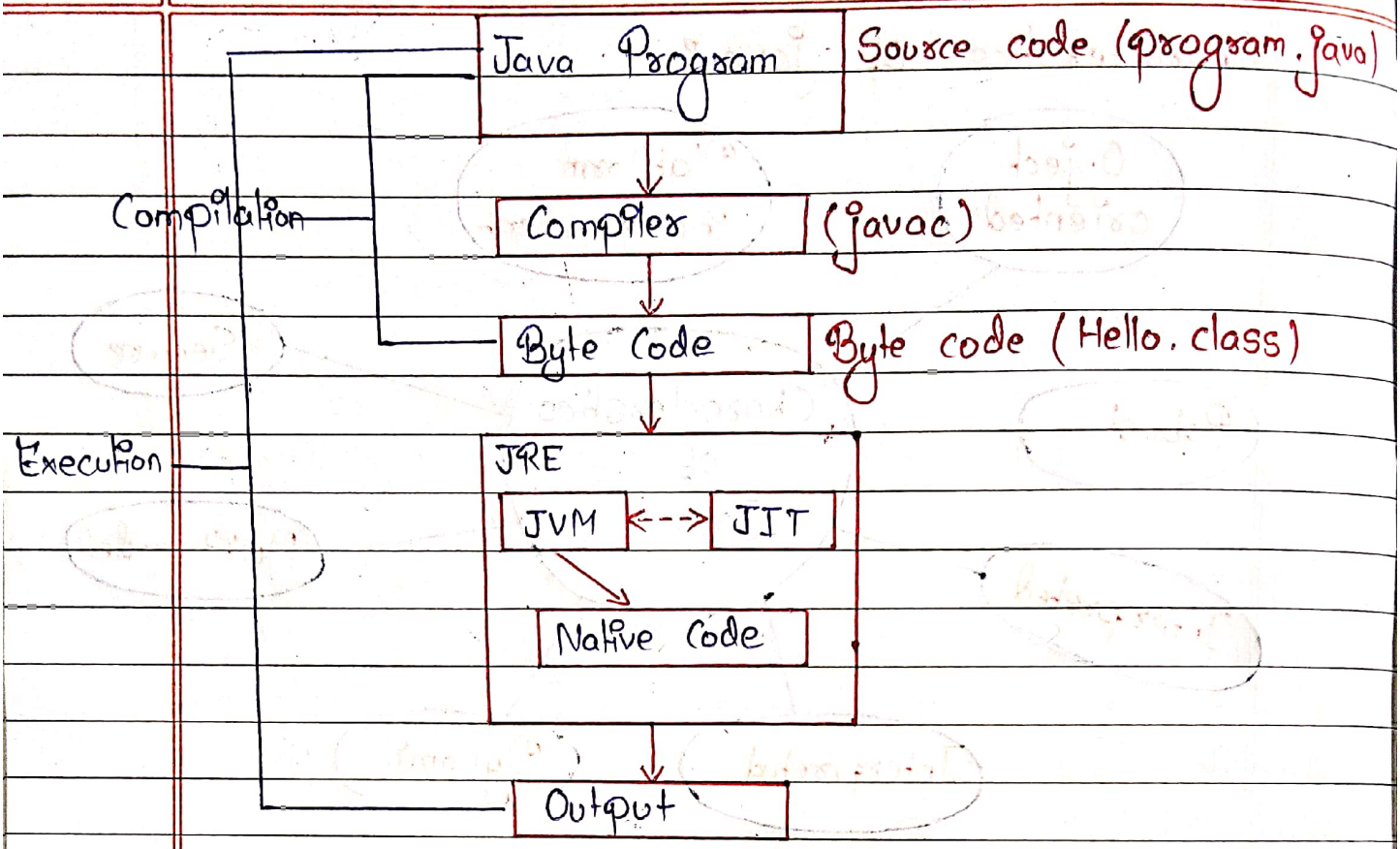
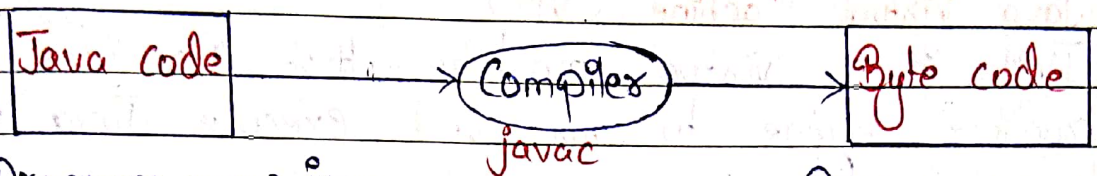


Fig: Compilation and Execution process of a java program

1. Creation of a java program :- Write the code of program in java on any text editor and save it with 'program_name.java' where program_name is the valid name of a java program and '.java' is the extension of a java program.

2. Compiling a java program :- After writing the java program, compile the program in a command prompt or any other console. If there are no errors in program, we can run it and get the desired output.



Program_name.java Program-name.class

3. ~~Loading the progr~~

3. Byte code formation :- After compilation, java compiler (javac) generates a byte code file. Byte code is a platform independent code of java program.

4. Verification for bytecode :- The bytecode verifies the code and it makes sure that the bytecodes are valid and accessible.

5. Java Program execution :- Now, the JVM (Java Virtual Machine) interprets the bytecode JVM is found inside JRE. JVM generates the Native code using JIT (Just-in-time). ~~whi~~ JIT helps to execute the program faster and it allows multiple programs execution at the same time.

When the Native code is generated which is machine language, JVM executes the program and displays the output.

Page No. :
Date :

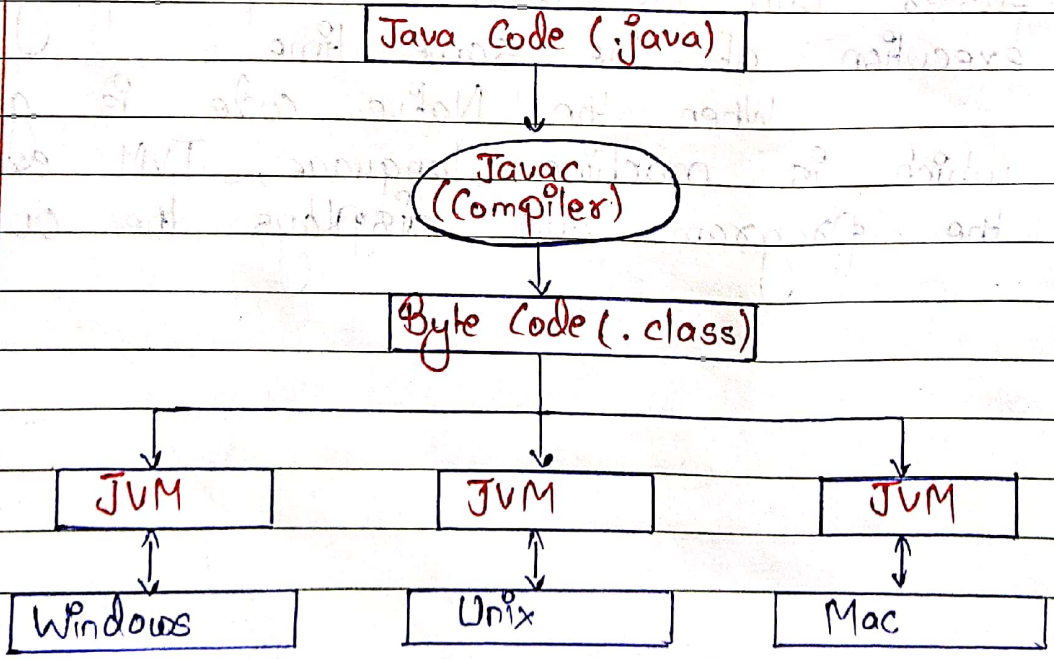
Java Virtual Machine (JVM)

* JVM is a virtual machine that provides the runtime engine to run and execute Java program.
* JVM converts byte code into Native code (Machine code).
* JVM allows to run the class file into multiple platform.
* JVM is a part of JRE.

How JVM works ?

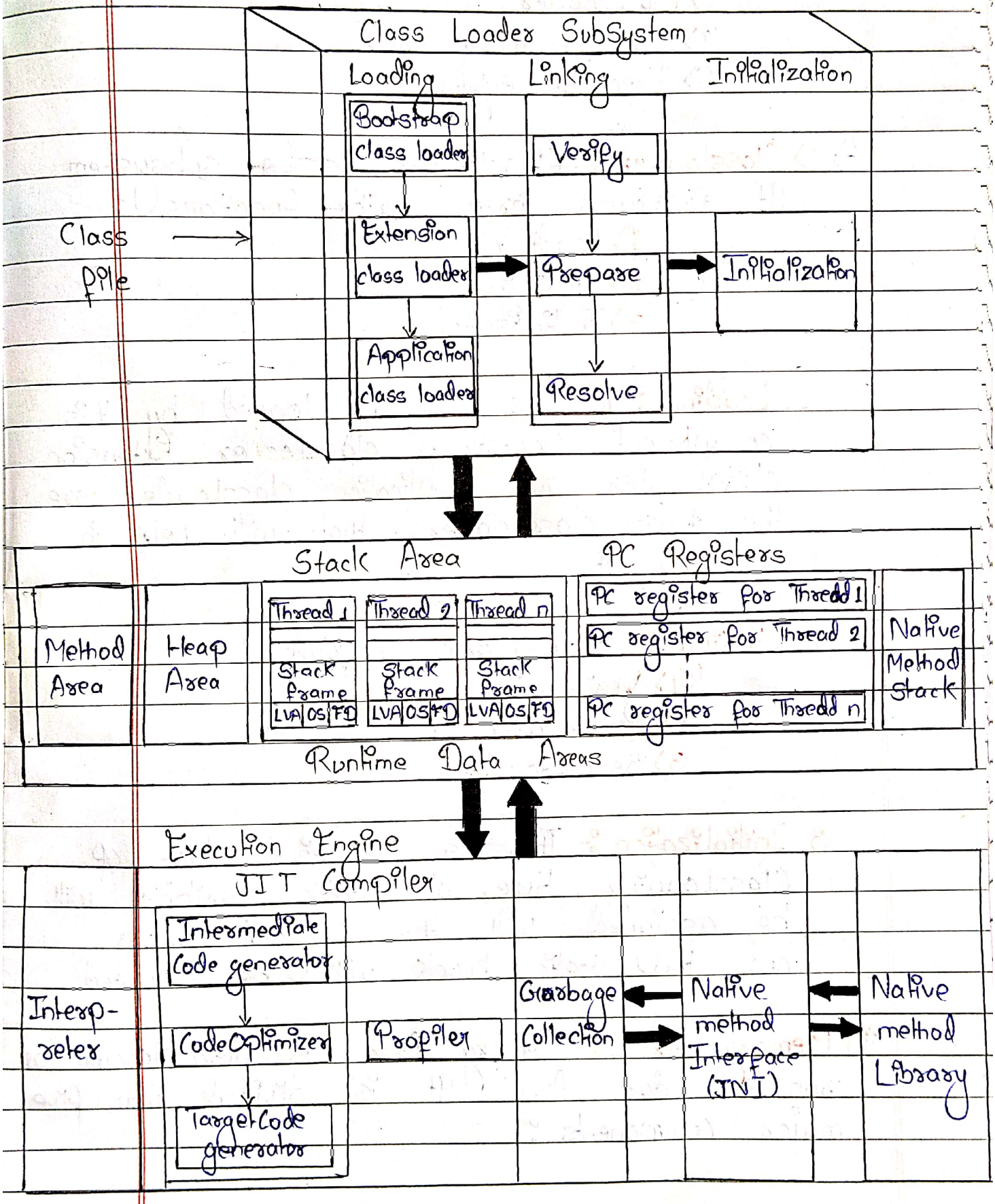
- Java code is compiled into bytecode
- JVM converts byte code ⟶ machine code
- CPU executes the java program.

So, JVM converts byte code into machine code and makes it executable for the CPU.



Page No. :
Date :

Architecture of JVM :



JVM architecture contains three main units :

i) Classloader
ii) JVM memory area
iii) Execution engine

i) Classloader :- Classloader is a subsystem. It performs three major functions :

1) Loading
2) Linking
3) Initialization

1. Loading :- Classes will be loaded by this component. Bootstrap classLoader, Extension classLoader and Application classLoader are the three classLoaders that will help in achieving it.

2. Linking :- It performs following :

1) Verification
2) Preparation
3) Resolution

3. Initialization :- This is the final stage of ClassLoading. Here, all static variables will be assigned with the original values and the static block will be executed.

ii) Memory Area :- Memory Area is also known as run-time data Area. It is divided into five major components :

Page No. :
Date :

1. Method Area
2. Heap Area
3. Stack Area
4. PC registers
5. Native method stacks

} common & shared across multiple threads

1. Method Area :- All the class level data will be stored here including static variables.

2. Heap Area :- All the objects, their related instance variables and arrays are stored in heap.

3. Stack Area :- Stack Area store local variables and its partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked and it is deleted when method invocation process is completed.

4. PC registers :- Each thread will have separate PC registers to hold the address of current executing instruction. Once the instruction is executed, the PC register will be updated with the next instruction.

5. Native Method stacks :- It holds the Native method information. For every thread, a separate native method stack will be created.

iii) Execution engine :- The bytecode which is assigned to the Runtime Data Area will be executed by the execution engine. The execution engine reads the bytecode and executes it piece by piece.

It has following components :

1. Interpreter
2. JIT Compiler – (Intermediate code generator, Code Optimizer, Target code generator, Profiler)
3. Garbage Collector
4. Java Native Interface (JNI)
5. Native Method Libraries

JRE (Java Runtime Environment)

- JRE is a software package that consists of JVM and other components inside it.
- Developed by Sun Microsystems.
- JRE provides runtime environment to run java progx

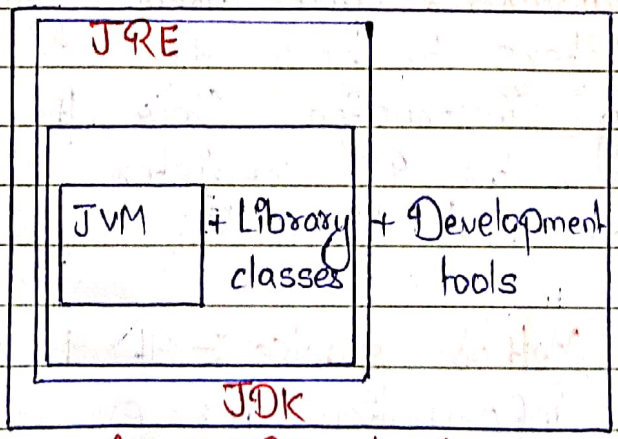
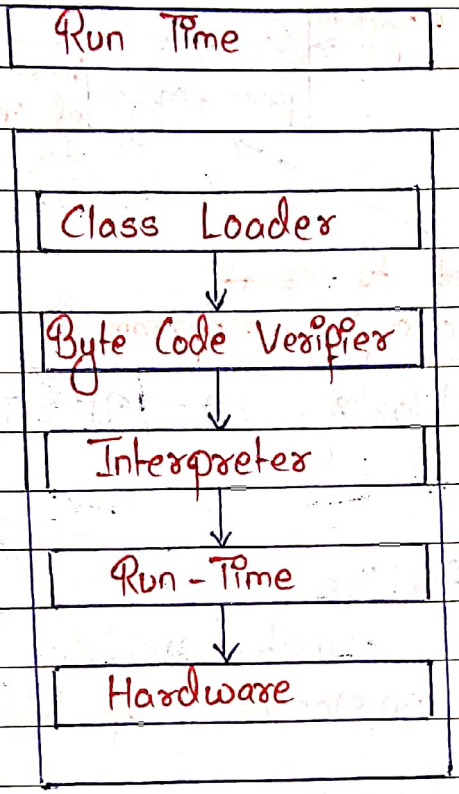


Fig : JRE structure

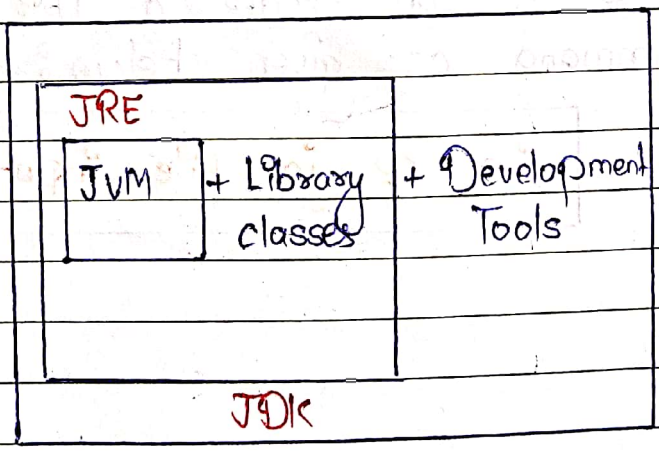
Page No. :
Date :

Working of JRE :



JDK (Java Development Kit)

- JDK is a kit or a set of tools which is used by the developers to execute, compile and test java programs.
~~it is~~



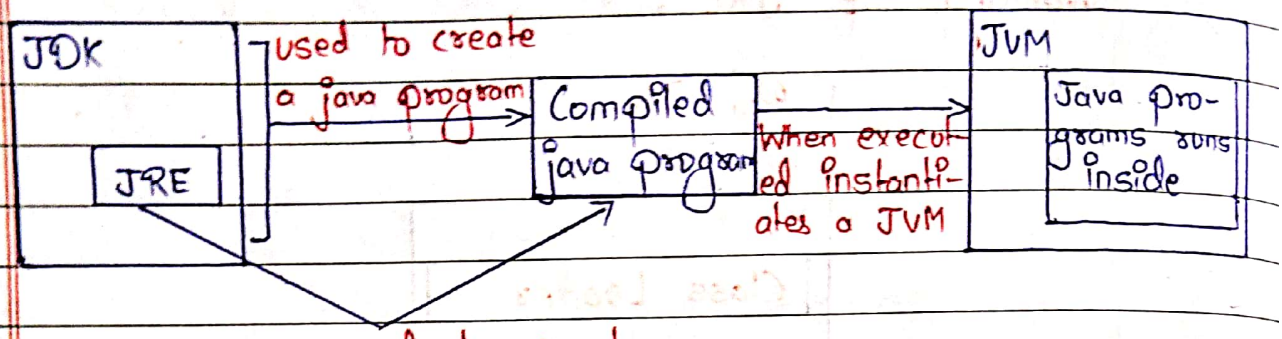


Fig: Working of JRE, JDK and JVM

JAR format

- JAR → Java Archive
- JAR is a file format / archiving tool which contains all the components of an executable java application.
- All the predefined libraries are available in this format.

We can create a JAR file using the command line options or using any IDEs.

Creating a JAR file:

We can create a JAR file using the jar command as given below:

```
jar cf jar-file input-file(s)
```

Page No. :
Date :

Platform Independency :- Java is platform independent Language. It runs in every platform like Android, Windows, Linux and Mac. Programs written in java can be run in any operating system, such as if java program is written in windows OS, then we will be able to run it easily in Linux OS as well.

Portability :- Java is a portable language because the source code of java is converted into Byte code with the help of compiler, this Byte code can be run in any system so it can be easily obtained.

Security :- Java is the most secure language because the java program runs in the JRE, before generating the machine code, the program detects the error by running some tests on the JVM. The java language is virus free which keeps the programs safe.

Page No. :
Date :

Naming conventions :

Identifiers	Naming Rules	Examples

Class	- Should not start with the uppercase letter.
	- It should be noun.
		public class Employee {
		//code
		}

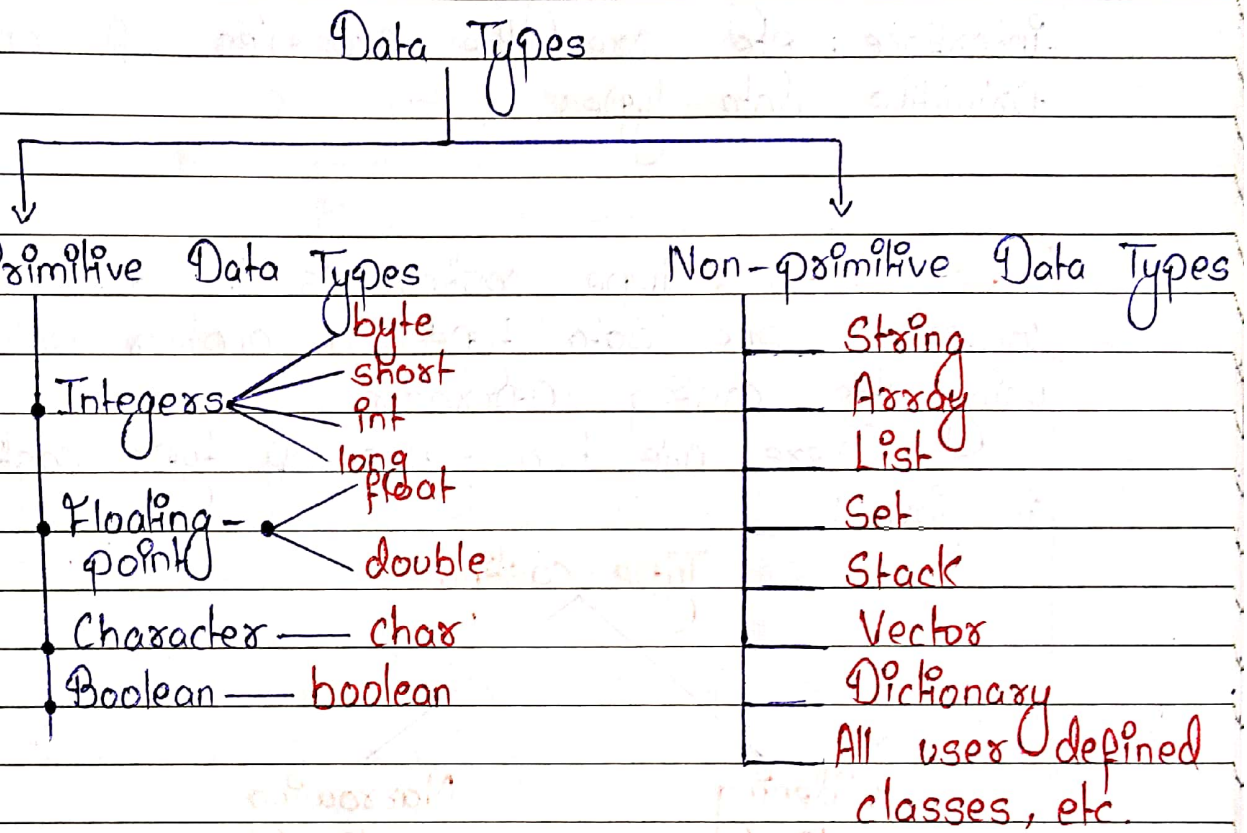
Interface	- Should start with upper-case letter.
	- Should be an adjective such as Runnable, Remote, etc.
		interface printable {
		// code snippet
		}

Method	- Should start with lowercase.
	- Should be a verb (main(), print(), println())
	- If the name contains multiple words start it with a lowercase followed by an uppercase like actionPerformed().
		class Employee
		{
		// method
		void draw()
		{
		// code
		}
		}

Page No. :
Date :

Data types in java :- Data type is a type of data which is stored in variables.

In java, data types are classified into two types and they are :



Primitive data types :- Primitive datatype is pre-defined datatype and it uses a reserved keyword.

Data type	Meaning	Memory size
byte	Whole numbers	1 byte
short	Whole numbers	2 bytes
int	Whole numbers	4 bytes
long	Whole numbers	8 bytes
float	Fractional numbers	4 bytes
double	Fractional numbers	8 bytes
char	Single characters	2 bytes
boolean	unsigned char	1 bit

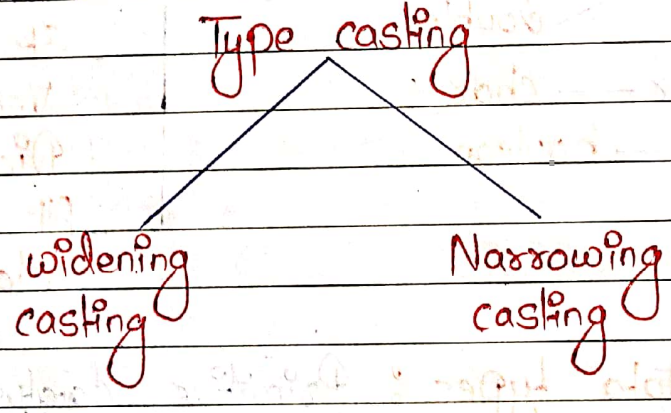
Page No.:
Date:

Non-primitive data types :- Non-primitive datatypes are the data types created by user. These are implemented using objects. Its default value is null.

String, Array, List, Queue, Stack, class, interface, etc. are the examples of non-primitive data types.

Type casting :- Type casting is the process of converting one data type to another data type using the casting operator.

There are two types of type casting in java :



Widening casting :- Converting a smaller type data to a larger type size. It is done automatically

byte → short → char → int → long → float → double

Narrowing casting :- Converting a larger type to a smaller data type. It is done manually.

double → float → long → int → char → short → byte

Page No. :
Date :

Operator :- An operator is a symbol used to perform arithmetic and logical operations.

1. Arithmetic operators
2. Relational (or) comparison operators
3. Logical operators
4. Assignment operators
5. Bitwise operators
6. Conditional operators

1. Arithmetic operators :- used to perform basic mathematical operations.

Operator	Function	Example
+	Addition	10+5 = 15
-	Subtraction	10-5 = 5
*	Multiplication	10*5 = 50
/	Division	10/5 = 2
%	Modulus	5%2 = 1
++	Increment	a++ or ++a
--	Decrement	a-- or --a

2. Relational operators :- Relational operators are the symbols that are used to compare two values or to check the relationship between two values. It returns either True or False values.

Operator		Example
<	→	10 < 5 is FALSE
>	→	10 > 5 is TRUE
<=	→	10 <= 5 is FALSE
>=	→	10 >= 5 is TRUE
==	→	10 == 5 is FALSE
!=	→	10 != 5 is TRUE

3. Logical Operators :- Used to combine multiple conditions into one condition. It also gives TRUE or FALSE values.

Operator	Function	Example
Logical AND (&)	~~Logica~~ TRUE if all conditions are true otherwise FALSE	false & true => false
Logical OR (|)	FALSE if all conditions are false otherwise returns TRUE	false | true => true
Logical XOR (^)	FALSE if all conditions are same otherwise returns TRUE	true ^ true => false
Logical NOT (!)	TRUE if condition is FALSE and returns FALSE if it is TRUE.	! false => true
short-circuit ~~(AND)~~ AND (&&)	Similar to Logical AND (&) but once a decision is finalised (meaning) it doesn't evaluate	false && true => false
short-circuit OR (||)	Similar to logical OR (|) but once a decision is finalised it doesn't evaluate meaning	false || true => true

Page No. :
Date :

4. Assignment operators :- Used to assign right-hand side variable to the left-hand side variable

Operator	Function	Example
=	Assign right-hand side value to the left-hand side variable	A = 15
+=	Add both sides value and store the result to the left-hand side variable	A += 10
-=	Subtract right-hand side value from left-hand side variable value and store the result in left-hand variable	A -= B
*=	Multiple right " " "	A *= B
/=	Divide left hand side variable with right-hand side variable value and store " " "	A /= B
%=	" " "	A %= B
&=	Logical AND assignment	-
|=	Logical OR assignment	-
^=	Logical XOR assignment	-

5. Bitwise operators :- Used to perform bit-level operations. It is performed on binary values

Operator	Function	Example
&	Returns AND if all bits are 1 or.	A & B => 16 (10000)
|	Returns 0 if all bits are 0.	A | B => 29 (11101)
^	Returns 0 if all bits are same	A ^ B => 13 (01101)
~	returns once complement negation of bit (flipping)	
<<	Shifts all bits to the left by specified no.	A << 2 => 100 (1100100)
>>	Shifts all bits to the right by specified no. of positions.	A >> 2 => 6 (00110)

6. Conditional operator : Also called Ternary operator. It requires three operators. Used for decision-making.

Syntax :

Condition ? TRUE part : FALSE : part

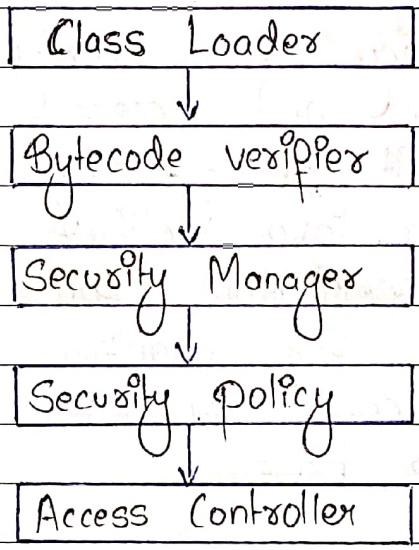
Security promises of JVM :

- Every object is constructed exactly once before it is used.
- Every object is an instance of exactly one class, which does not change through the life of the object.
- If a field or method is marked private, then the only code that ever accesses it is found within the class itself.
- Fields and methods marked protective are used only by code that participates in the implementation of class.
- Every local variable is initialized before it is used.
- It is impossible to underflow or overflow the stack.
- It is impossible to read or write past the end of an array or before the beginning of the array.
- It is impossible to change the length of an array once it has been created.
- Final methods cannot be overridden and final classes cannot be subclassed.

Page No. :
Date :

Security Architecture :- The security architecture in java is designed to provide a secure execution environment for java applications by controlling and managing access to system resources, protecting data integrity and ensuring code authenticity.

It consists of various components and mechanisms including :



1. **Class Loader :-** The class Loader is responsible for loading classes into the JVM. It plays a crucial role in the security architecture by enforcing class loading restrictions, ensuring that only trusted classes are loaded.

2. **Bytecode verifier :-** Before executing java bytecode the JVM verifies the bytecode to ensure that it adheres to certain safety constraints, preventing the execution of potentially harmful code.

3. Security Manager :- The security manager is a part of the java security Architecture that manages the security policy. It controls access to system resources such as file I/O, network access and native code execution. If an application tries to perform an action that is not allowed by the security policy, the security manager denies it.

4. Security policy :- The security policy defines the permissions and access rights for java applications. It is typically specified in a policy file (eg. java.policy) and dictates what actions an application can perform. The policy includes rules for granting or denying permissions to code sources, classes and individual code bases.

5. Access controller :- The access controller is responsible for enforcing the security policy by checking permissions for various code sources and classes. It ensures that the code executed within the JVM complies with the defined security rules.

Security policy :- The security policy in java is a critical part of the security architecture and dictates what actions are allowed for java applications. It is typically defined in a policy file, which is written in a specific

format. The policy file contains entries that specify the following :

1. Code sources :- These entries define the sources of code, which can be URLs, JAR files or directories.

2. Permissions :- Permissions are associated with code sources, and determine what actions are followed allowed for that code. Common permissions include file I/O access, network access and reflective operations.

3. Principals :- Principals are used to specify who is executing the code and can be used for more fine-grained permission control.

4. Signed JAR files :- The policy can specify whether code from digitally signed JAR files should be granted additional privileges.

Page No. :
Date :

JDBC and JDBC Drivers

* JDBC (Java Database Connectivity) is an API for connecting and interacting java with databases.

* JDBC drivers are s/w components that provides th necessary functionality to connect java applications to different types of databases.

There are four types of JDBC drivers :

1. Type 1 - ~~JB~~ JDBC - ODBC bridge driver
2. Type 2 - Native API partly java Driver
3. Type 3 - Network protocol driver
4. Type 4 - Thin Driver (also known as the Direct to database Pure driver)

Each type of driver has its own advantages and is suitable for different scenarios.

1. Type - 1 : JDBC - ODBC bridge driver
→ Oldest Driver.
→ Written in C language.
→ Scalability issues.
→ Outdated so, not in use now.
→ Performance issues & not so efficient.

database vendors are — MySQL, oracle, IBM, etc.

Page No. :
Date :

2. Type-2 : Native API partly Java Driver
→ Provides database vendors.
→ Database vendors are database providing companies like MySQL, oracle, IBM, etc. These companies also provides their own API.
For ex : MySQL provides MySQL API, Oracle provides oracle API & we are bound to use that same company 's API if we are using the database of a company. This was the problem of Native API driver.

3. Type-3 : Network Protocol driver
→ Better than JDBC-ODBC driver & Native API driver.
→ It connects the database with client but through a middleware. This was the only issue of Network protocol driver.

4. Type-4 : Thin-Driver
→ Connects database & client directly.
→ No middleware.
→ No any performance & scalability issue.
→ Pure written in pure java.
→ Directly converts native API calls in database calls.
→ Light-weight.
→ Much efficient.
→ Most used driver nowddays.

Page No. :
Date :

JDBC Components : In addition to the JDBC drivers there are several other components that make up the JDBC API, including :

- DriverManager class
- Connection interface
- Statement and PreparedStatement interfaces
- ResultSet interface

DriverManager class is the only class component & all other components are its interfaces.

DriverManager class
→ DriverManager class has various methods. Among them, 'getConnection' is the most important & widely used method.

DriverManager class helps to connect with database through '.getconnection'.

Connection interface
It provides methods such as close(), commit(), rollback(), createStatement(), prepareCall(), prepareStatement(), etc. to represents the connection with database.

Statement & preparedStatement interfaces
Statement interface is used to run statement queries in java.

It is of types i.e. Statement interface &

Page No. :
Date :

PreparedStatement interfaces.

ResultSet interface

The output which we received when we run queries in java is in the form of stored in an instance and that instance is created by ResultSet interface.

These components work together to provide a powerful & flexible API for working with databases in java.

~~Data~~

~~Database connectivity using JDBC :~~

Basic Database operation :

Page No.:
Date:

Threads in java : Thread is a light-weight process that allows us to perform multiple tasks independently.

A thread is the smallest unit of execution within a process.

Life Cycle of a Thread

⇒ During the lifetime of a thread, it passes through many states. The different states of the thread are as follows:

1. Newborn State
2. Runnable State
3. Running state
4. Blocked state
5. Dead State

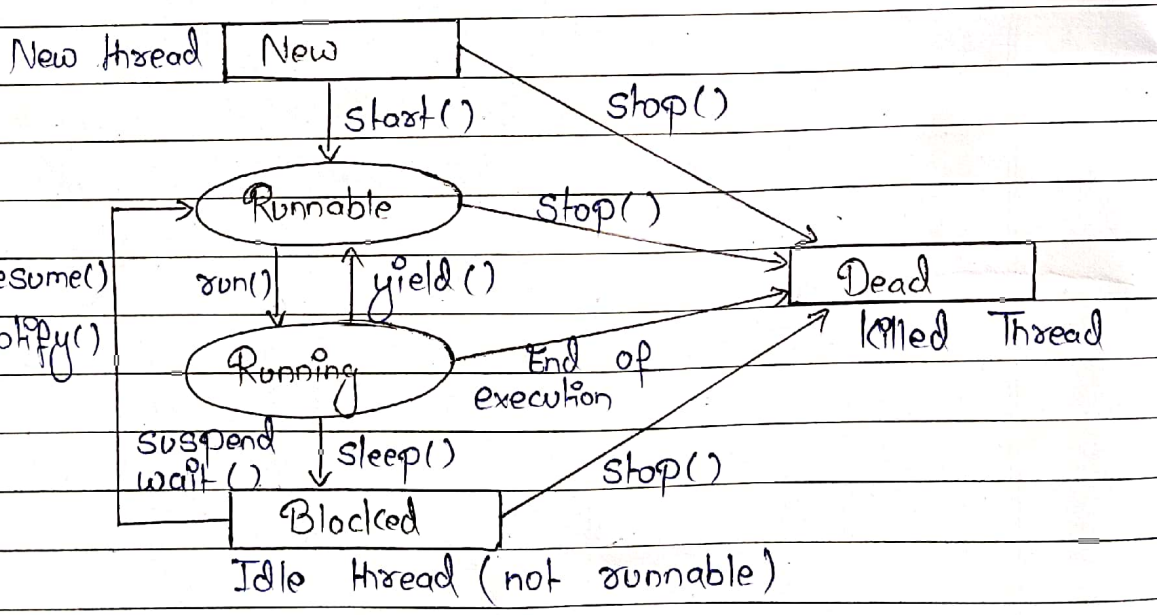


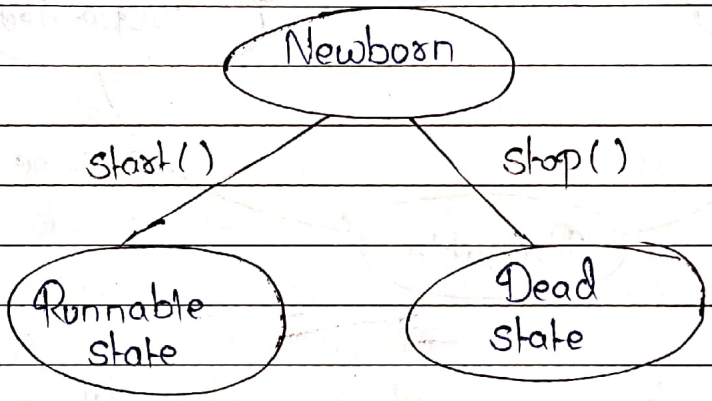
Fig : Lifecycle of a thread

1. Newborn State :- When a thread object is created, the thread is said to be in Newborn state. At this moment, the thread is not scheduled for running.

At this state, the following may be done:

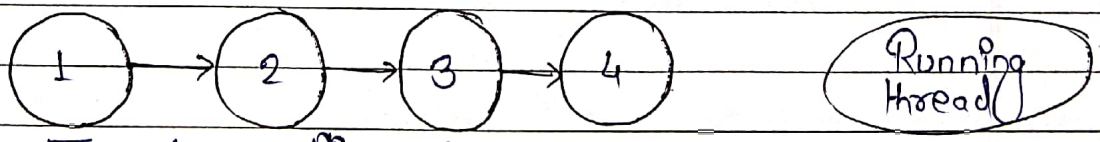
i. We can ~~call~~ run the thread by calling start() method.

ii. We can kill the thread by using stop() method.



2. Runnable state :- The runnable state means that the thread is ready for the execution and is waiting for the availability of the processor. In simple terms, the thread is ready but has not got the processor time.

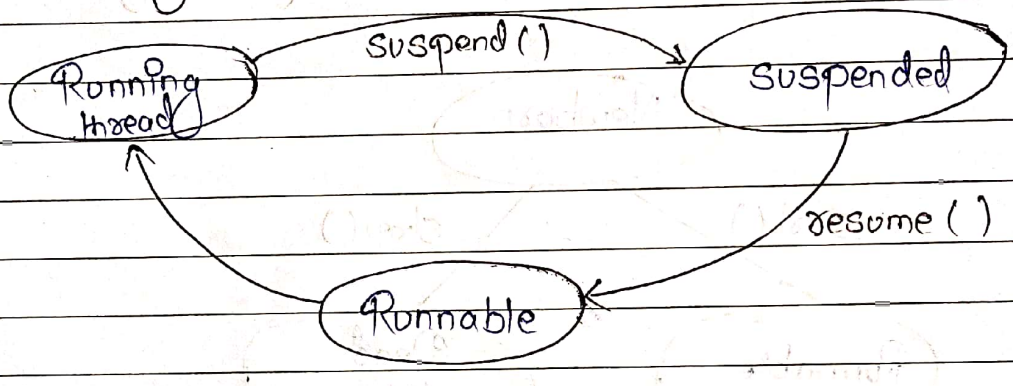
All the thread has equal priority & hence all follow a queue system & the processor gives time to each thread in round robin system.



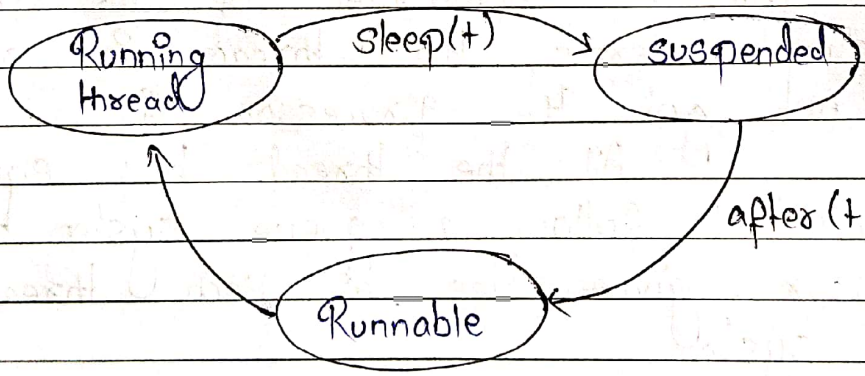
3. Running state :- Running state means that the processor has given its time to the thread for its execution.

A running thread may have following conditions :

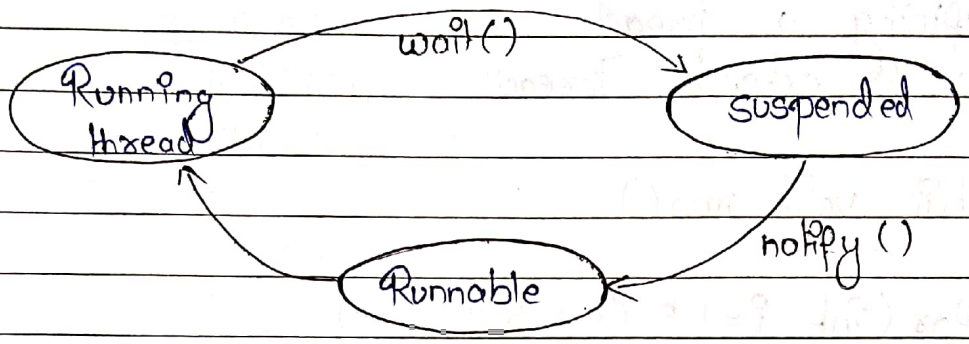
a. It has been suspended using suspend() method. A suspended thread is revived by using resume() method.



b. It has been to sleep using sleep(time) method where time is in miliseconds. After the given time, the thread itself joins the runnable state.



c. It has been told to wait until some event occurs. This is done using wait() method & it is rescheduled to run again by notify() method.



4. Blocked state :- A thread is said to be in blocked state when it is prevented from entering into the runnable state & subsequently the running state. This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.

5. Dead state :- A thread is said to be in dead state when it has been killed by using ~~kill()~~ stop() method or it has successfully completed its execution.

Creating thread : We can create / define a thread in java in two ways :

1. By extending thread class
2. By implementing runnable interface

1. By extending Thread class :- A thread class name "Thread" is already defined in java. We just need to extend it through a class.

```
// Defining a thread
class A extends Thread
{
  public void run()
  {
    for(int i=1; i<=5; i++)
    {
     System.out.println("Akhilesh");
    }
  }
}
// creating main thread and calling child thread in it
class B
{
  public static void main(String[] args) {
     A t = new A(); // creating instance of child
     t.start();          thread class A

    for(int i=1; i<=5; i++)
    {
    System.out.println("Ankush");
    }
  }
}
```

Page No. :
Date :

Output :

Ankush
Ankush
Alkhilesh
Ankush
Alkhilesh
Alkhilesh
Ankush
Alkhilesh
Ankush
Alkhilesh

2. By Implementing Runnable interface :- In this technique, we need to implement the runnable interface. Runnable interface contains a method 'run' in it.

```
// Defining a thread through runnable interface

class A implements Runnable
{
  public void run() // method of runnable interface
  {
    // code; // task to be performed by thread is defined here.
  }
}

// Creating another class which starts above thread
class B
{
  public static void main(String [] args);
  {
```

Runnable interface has only one method i.e. run(), it doesn't have start() to start the method. Thus, to start thread in runnable interface we create the object of thread class, Since, thread class contains start() method in it.

```
A r = new A();  // creating object of class A
Thread t = new Thread();
Thread (r); // object of run() should be passed as
t.start();      parameter in thread object
}               to execute run() method's co
}
```

Program :

```
class A implements Runnable
{
 public void run()
 {
   for (int i=1; i<=5; i++)
   {
   System.out.println("My Child Thread");
   }
 }
}

class B
{
 public static void main (String[] args) {
     A r = new A();
     Thread t = new Thread (r);
     t.start();
 }
}
```

Main thread's job:
1. To create object of child thread & to start the child thread.
2. To execute the code of main thread.

Job of child thread is to execute the code of child thread.

```
for (i=1; i<=5; i++)
{
  System.out.println("Main thread class");
}
}
}
```

Output:

main thread
My child thread
My child thread
My child thread
My child thread
My child thread
main thread
main thread
main thread
main thread

Thread Priority

=> In java, by default all the threads have equal priority & hence they share the processor on a first-come-first-serve basis. But, we can assign priority to thread using these two methods:

a. setPriority()

b. getPriority()

a. setPriority() :- Java permits us to set the priority of a thread using the setPriority() method.

The syntax is :

ThreadName.setPriority(intNumber)

Here, intNumber is an integer value to which the thread's priority is set. The Thread class provides us many thread priority constants :

MIN_PRIORITY = 1
NORM_PRIORITY = 5
MAX_PRIORITY = 10

Here, the intNumber may be set to any priority level from 1 to 10 denoting least priority, 10 denoting highest priority.

Program :

```
class A extends
{
public void run()
{
System.out.println("Thread A started");
for (int i=1; i<=5; i++);
{
System.out.println("From Thread A : i=+i);
}
System.out.println("Thread A Ends Here---");
}
}
```

```
Class B extends Thread
{
  public void run()
  {
  System.out.println("Thread B is started");
  for(int j=1; j<=5; j++)
  {
  System.out.println("From Thread B: j="+j);
  }
  System.out.println("Thread B Ends here---");
  }
}

Class C extends Thread
{
  public void run()
  {
  System.out.println("Thread C is started");
  for(int k=1; k<=5; k++)
  {
  System.out.println("From Thread C: k="+k);
  }
  }
  System.out.println("Thread C Ends Here---");
  }
}
```

```
class ThreadPriority
{
 public static void main(String[] args)
 {
    A aa = new A();
    B bb = new B();
    C cc = new C();
    cc.setPriority(Thread.MAX_PRIORITY);
    bb.setPriority(aa.getPriority()+1);
    aa.setPriority(Thread.MIN_PRIORITY);
    System.out.println("Thread A is started--");
    aa.start();
    System.out.println("Thread B is started--");
    bb.start();
    System.out.println("Thread C is started--");
    cc.start();
    System.out.println("Main Thread Ended---");
 }
}
```

b. getPriority() :-

Program :-

Thread Priority, Thread ID

```
class ThreadID
{
 public static void main
```

```
class A extends Thread {
    public void run()
    {
        System.out.println("\n Inside thread A: \nTh-
        read ID: \t" + Thread.currentThread().getId()+
            "\n ThreadPriority: \t" + Thread.currentThread().
                getPriority());
    }
}
```

Thread Synchronization

⇒ We know that threads uses their own data and methods provided inside the run() method. Consider a case when they try to use data & methods outside themselves. In such situation, they may compete for the same resource & may lead to serious problems.

For instance, suppose there are two threads, one they may reading the data from the resource & the second they may be writing the data at the same time. In such situation, result will be unexpected. Java enables us to overcome this situation using a technique called Synchronization.

Synchronization: The keyword synchronized helps to solve problem by keeping watch on such locations. To avoid issues, the method that will read information may be declared as synchronized.

Page No. :
Date :

for ex :

```
Synchronized void write()
{
--------;
--------;
}
Synchronized void read()
{
------;
------;
}
```

When we declare a method as synchronized, java creates a "monitor" & hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code.

Page No. :
Date :

Thread communication :- When a thread communicates with another thread by sending some signals to ask a thread to wait or wakes up another thread from sleep, it is done by interthread communication.

Note : The thread should be inside the synchronized Area.

Thread communication is achieved using three methods :

i. wait()
ii. notify()
iii. notify all()

i. wait() :- It tells the thread to give up the monitor & go to sleep until some other thread enters the same monitor & calls notify() or notify all().

ii. notify() :- wakes up a thread that called wait on the same object.

iii. notify all() :- It wakes up all the threads on which wait() was called on the same object.

Page No. :
Date :

Program :

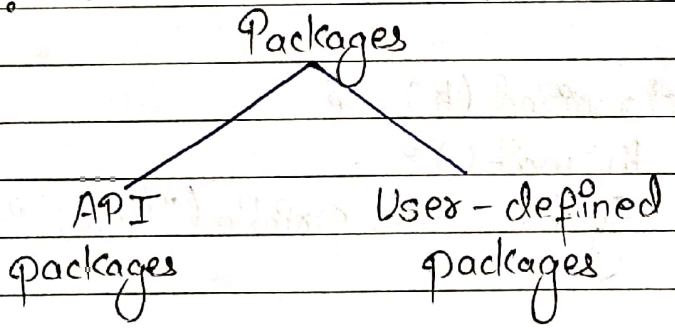
```
class SumDemo extends Thread {
    int sum = 0;

    public void run()
        synchronized (this) {
            for (int i=1; int i<=5; i++) {
                sum = sum + i;
            }
            this.notify();
        }
    }
}
public class ThreadCommunicationDemo {
    public static void main(String[] args) throws
                                    InterruptException {
        SumDemo th = new SumDemo();
        th.start();

        Synchronized (th) {
            th.wait();
            System.out.println("Sum is " + th.sum);
        }
    }
}
```

Page No. :
Date :

Packages :- A ~~ba~~ package is basically collections of different classes. In other words, we can say that package is a way to group variety of classes &/or interfaces together. The grouping is usually done on the basis of functionality.

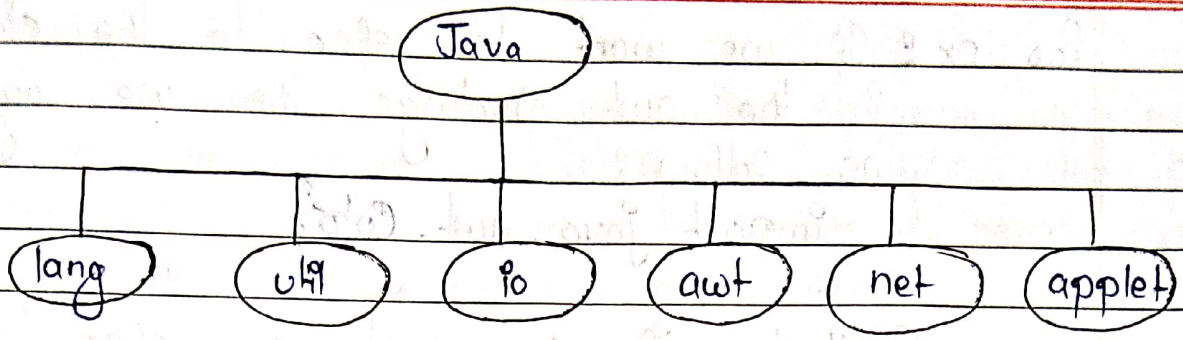
Need of packages :
- The class contained in a package can be used in any program that import that classes.
- Packages provide a way to "hide" classes.
- Package also provides a way for separating design from coding.

Types of packages :
Packages are broadly categorized into two ways :



Java API package :- Java API (Application Programming Interface) packages provides a way to group different classes into different packages. They are built-in packages.

The different Java API packages (built-in) packages are :-



java.lang - Language support classes. These are classes that java compiler itself uses & therefore they are automatically imported. They include classes for primitive data type, string, maths functions, threads & exceptions.

java.util - language utility classes such as vector, random number, scanner, etc.

java.io - input/output support classes

java.awt - Set of classes to implement GUI

java.net - Class for networking

java.applet - Classes for creating & implementing applets.

Associating classes to packages

=> There are two ways to access the classes stored in the packages. The first approach is to use the fully qualified class name of the class that we want to use.

Page No. :
Date :

For ex: If we want to refer to the class Color in the awt package, then we may do as follow:

```
import java.awt.Color
```

Similarly, if we want to refer to the Scanner class in the util package, then we may do as follows:

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

Another way to implement packages in java through the naming conventions. Packages can be names using the standard java naming rules By convention, however, packages being with lowercase letters. This makes it easy for users to distinguish package name from class name.

```
double y = java.lang.Math.sqrt(x);
```

Here, lang is the package, Math is the class & sqrt is the function i.e. the method name.

Page No. :
Date :

Exception handling

=> When we write program, we may make some common mistakes while typing the program to produce unexpected results. Errors are the wrongs that can make a program to go wrong.

Due to errors, the program may not work properly & hence the errors found & rectified

The errors are broadly categorized into :

1. Compile time errors
2. Run time errors

1 Compile time errors :- All syntax errors that can be detected & displayed by the java compiler are called compile time errors. Whenever, our compiler shows an error, we rectify them & again re-compile the whole code

Ex :

```
class abc
{
public static void main(String [] args) {
{
System.out.println("Hello Java") //semi-colon missing
}
}
```

2. Run-time errors :- Sometimes it happens that our program compiles successfully but when we run, it gives us strange result that is unexpected result. This may be due to the wrong logic of the program & hence our program produces unexpected results.

The most common run-time errors are :

i. Divide by zero
ii. Accessing an element beyond index in an array
iii. The file that we need, do not exists
Storing wrong type of values in array
& many more ---

Ex :

```
class abc
{
public static void main(String[] args) {
{
int a=10;
int b=5;
int c=5;
int x = a/(b-c);        // throws exception
System.out.println("X = " + x);
int y = a/(b+c);
System.out.println("y = " + y);
}
}
```

In line 8 of the above code, we have written $x = a/(b-c)$ & this line ~~gives exce~~ throws an exception because when we divide any number by 0, it is undefined. Thus, the compiler throws unexpected error.

Unexpected error is actually the compile time errors. Whenever an unexpected error occurs in our code, the program gets terminated there Although, the remaining ~~program~~ code has no any error. ~~but when~~

Exception :- An exception is a ~~run-time error~~ condition that is caused by the run-time error. When java interpreter encounters a run-time error which is divide by zero, it creates an exception object & throws it (i.e. inform about the error).

Now, if we have written the code to catch the exception & handle it properly, then our program will keep running, otherwise the whole program will collapse means terminate.

So, the process of catching the exception thrown by the error condition & then display an appropriate message for the same is called "exception handling".

The purpose of exception handling mechanism is to provide a means to detect & report a 'exceptional circumstance' so that appropriate action can be taken. The mechanism suggest incorporation of a separate error handling code that performs the following task:

1. Find the exception (Hit the exception)
2. Inform the error (Throw the exception)
3. Receive the error (Catch the exception)
4. Take action (Handle the exception)

Common Java Exception:

ArithmeticException - Caused by Maths errors such as divide by zero

ArrayIndexOutOfBoundsException - Caused by bad array index

ArrayStoreException - Caused by storing wrong data-type in an array

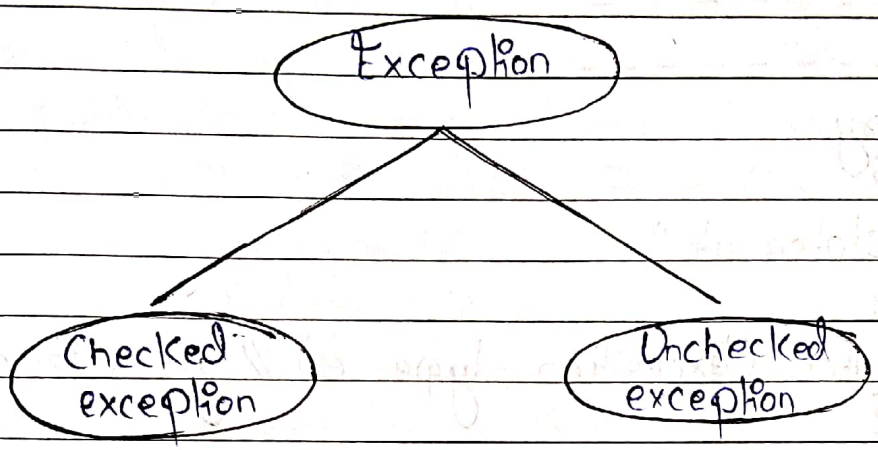
FileNotFoundException - Caused if the file do not exists

IOException - Caused by general I/O failures

StringIndexOutOfBoundsException - Caused by trying to find non-existent characters in String

Page No. :
Date :

Exception in java



1. Checked exception :- These exceptions are handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from java.lang.Exception.

2. Unchecked exception :- These exceptions are not essentially handled in the program code. The JVM handles such exceptions i.e. unchecked exceptions. Unchecked exceptions are extended from java.lang.RuntimeException class.

Exception handling :- Java uses a keyword called 'try' to preface a block of code that might contain some error. A catch block defined by the keyword "catch" catches the exception thrown by "try" block. The "catch" block is added immediately after the try block.

Page No. :
Date :

Syntax :

```
---------
---------
try
{
statement ;
}
catch (exception-type-e) // generate an exceptio
{
statement ;          // processes the exception
}
---------
---------
```

Program :

```
class demo
{
public static void main (String [] args)
{
int a=10, b=5, c=5, x, y;
try
{
x=a/(b-c); //
}
catch (ArithmeticException e)
{
System.out.println("Divide by zero");
}
y=a/(b+c);
System.out.println("y="+y);
}
}
```

Page No. :
Date :

Throw keyword :- This keyword is used to throw an exception explicitly.

Ex :

```
public class ExceptionDemo {(int age)
    static void canVote (String [] args) {
        int age ;
        if (age < 18)
        try {
            throw new Exception ();
        } catch (Exception e) {
            System.out.println("You are not in adult!");
        }
        else
            System.out.println("You can vote!");
        }
        public static void main (String [] args) {
            canVote (20);
            canVote (10);
        }
    }
```

Output :

```
You can vote!
You are not an adult!
```

Java Finally block :- Contains the code that must be executed no matter if an exception is thrown or not. It contains code of file release, closing connections, etc.

Page No. :
Date :

Ex :

```
class Main {
    public static void main (String [] args) {
    try {
        System.out.println(4/0);
    } catch (Exception e)
    {
        System.out.println(e);
    }
    finally
    {
    System.out.println("finally executed");
    }
        system.out.println("end");
    }
}
```

Output :

```
java.lang.ArithmeticException : / by zero
finally executed
end
```

Checked exceptions	Unchecked exceptions
Occur at compile time.	Occur at runtime.
The compiler checks for a checked exception.	The compiler doesn't check for exceptions.
Can be handled at compile time.	Can't be handled during compile time.
The JVM requires that the exception be caught & handled.	The JVM doesn't require the exception to be caught & handled.
Ex : File Not Found exception	Ex : No such element exception.

Page No. :
Date :

Final Keyword :- In Java, the final keyword is used to denote constants. It can be used with variables, methods & classes.

Once an entity (variable, class or method) is declared 'final', it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value.
- the final method cannot be overridden.
- the final class cannot be extended

i. final variable
ii. final class
iii. final method

i. final variable :- If a variable is declared final, its value cannot be changed during the execution of the program.

Ex :

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

    final int Age = 30;
    Age = 45;
    System.out.println("Age: " + Age);
    }
}
```

Output :

cannot assign a value to final variable Age
Age = 45;
^

ii. Final method :- If a method is declared final, it cannot be overridden throughout the execution of a program.

Ex :

```
class FinalDemo {
    public final void display() {
        System.out.println("This is a final method.");
    }
}
class Main extends FinalDemo {
    public final void display() {
        System.out.println("The final method is
                            overridden.");
    }

    public static void main(String[] args) {
        ~~System.out.println(~~
        Main obj = new Main();
        obj.display();
    }
}
```

Output :

```
display() in Main cannot override display()
in FinalDemo
    public final void display() {
                      ^
    Overridden method is final
```

iii. Final class :- If a class is declared final, the final class cannot be inherited by the another class.

Ex:

```
final class FinalClass {
    public void display() {
        System.out.println("This is a final
                            method");
    }
}

class Main extends FinalClass {
    public void display() {
        System.out.println("This final method
                            is overridden.");
    }

    public static void main(String[] args) {
        Main obj = new Main();
        obj.display();
    }
}
```

Output:

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

Page No. :
Date :

String : Usually string are the sequence of characters but in java, string is a class.

Ex : "hello" is ~~sequence~~ a string containing a sequence of characters 'h', 'e', 'l', & 'o'.

We use double quotes to represent a string in java.

Ex : String type = "Java Programming" ;

Here, we have created a string variable named 'type'. The variable is initialized with the string 'Java Programming'.

StringBuffer : StringBuffer is just like string in java but stringBuffer is mutable i.e. changes in it possible while string is immutable.

Creating stringBuffer :

StringBuffer sb = new StringBuffer ("Study");

StringBuffer is a class.

It provides various operations :

i. append ()
ii. replace ()
iii. reverse ()
iv. capacity ()
v. insert ()

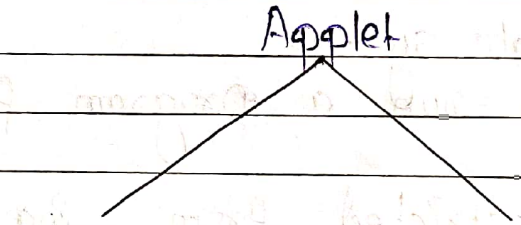
Page No. :
Date :

Applet

⇒ Applets are small java program that are primarily used in internet computing. They can be transported over the internet from one computer to another & run using the Applet Viewer or any web browser that support java.

It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation, etc.

Java has enabled interactive multimedia web documents. A web page now contain not only a simple text or static image but also a java applet which can produce graphics, sounds & moving images. Java applets therefore have begun to make a significant impact on the WWW.



Local applet :- An applet developed locally & stored in a local system is known as local applet. When a web page is trying to find a local applet, it doesn't use internet & therefore the local system does not require the internet connection. It simply searches the directories in the local system & locates & loads the specified applet.

Page No. :
Date :

Remote applet :- A remote applet is that which is developed by someone else & stored on a remote computer connected to the internet. If our system is connected to the internet, we can download the remote applet onto our system via the internet & run it. In order to load a remote applet, we must know the applet's address i.e. URL.

Applet v/s Application

1. Applet doesn't have main() method unlike applications.
2. Unlike standalone applications, applets cannot run independently. It needs web page using a special features known as html tag.
3. Applets cannot read from or write to the file in a local system.
4. Applets cannot run a program from local computer.
5. Applets are restricted from using libraries from other languages like C/C++, etc.

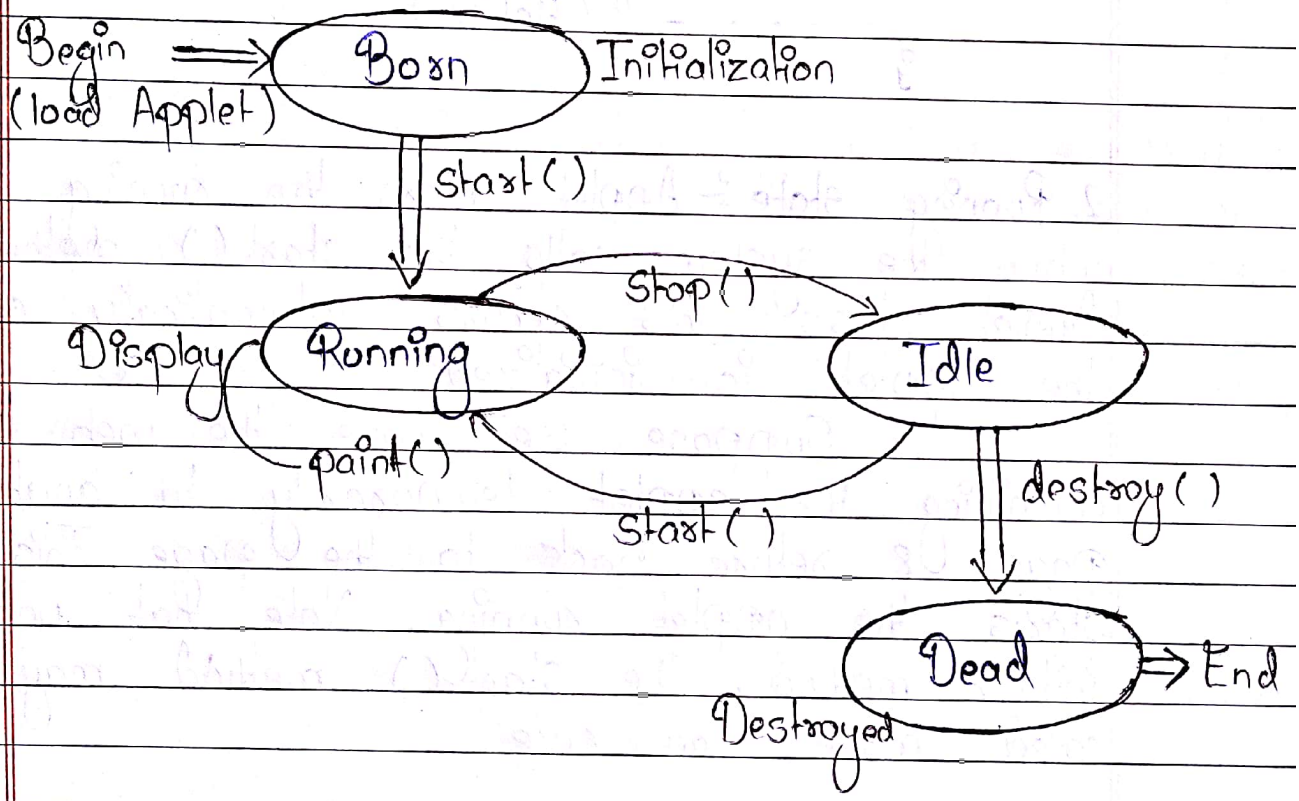
Applet Life Cycle

⇒ Every java applet has a life-cycle through which it passes throughout his life span. Every java applet inherits the behaviour from the applet class.

Page No. :
Date :

Applet lifecycle contains the following states:

1. Born or initialization state
2. Running state
3. Idle state
4. Dead or destroyed state



1. Initialization State :- When an applet is loaded, it enters the initialization state. This happens by calling the init() method. This init() method is called automatically. When init() method is called, the applet is born.

At this stage, the following actions may be done:

1. Create objects as required by the Applet
2. Set up initial values
3. Load images or fonts
4. Set up colors

Initialization occurs only once in the applet lifecycle. To go with any of the above point we need to override init() method :

```
public void init()
{
---------;
---------; (Action)
}
```

2. Running state :- Applet enters the running state when the system calls the start() method of Applet class. This occurs automatically after the applet is initialized.

Suppose, we leave the web page containing the applet temporarily to another page & return back to the page. This again starts the applet running. Note that, unlike init() method, the start() method may be called more than once.

```
public void start()
{
---------;
---------; (Action required)
}
```

3. Idle or stopped state :- An applet becomes idle when it is stopped from running. An applet is stopped automatically when we leave the current web page on which applet was

running. We can also stop the applet by calling the stop() method.

```
public void stop()
{
--------;
--------; (Action)
}
```

4. Dead state :- An applet is said to be dead when it is removed from the memory. This occurs automatically by invoking the destroy() method when we quit the browser. Like ~~initialization~~ initialization, destroying occurs only once in the applet's lipe cycle.

```
public void destroy()
{
--------;
--------; (Action)
}
```

5. Display state :- Whenever an applet has to perform some O/p operation, it enters display state. This happens after an applet enters to the running state. The paint() method is accomplish this task.

```
public void paint(Graphics g)
{
---------;
---------; (Action)
}
```

"Display state" is not actually considered as lipecycle pach.