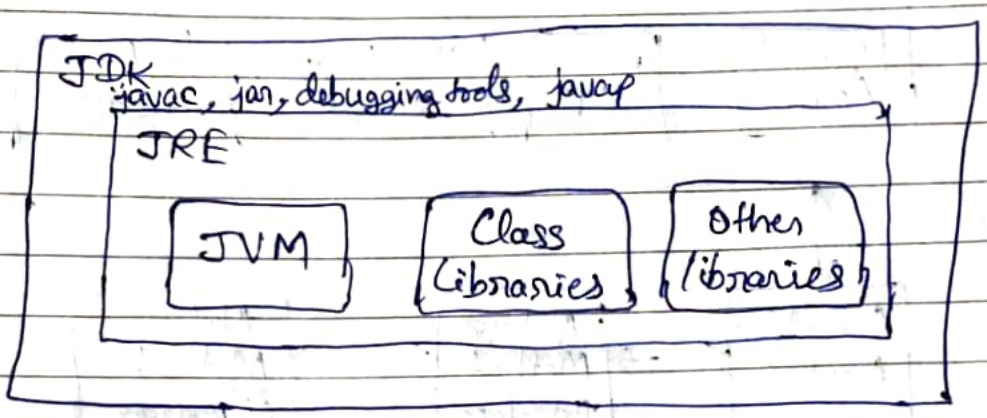
(1) JVM Architecture

DATE
PAGE

→ JVM acts as a run time engine to run Java applications JVM is the one that actually calls the main method present in Java code. JVM is a part of JRE (Java Runtime Environment).



→ Java applications are called WORA (Write Once Run Anywhere).

→ A programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

→ When we compile a .java file, .class files (contains byte code) with the same class name present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

Class Loader

DATE
PAGE

Class Loader Subsystem :-

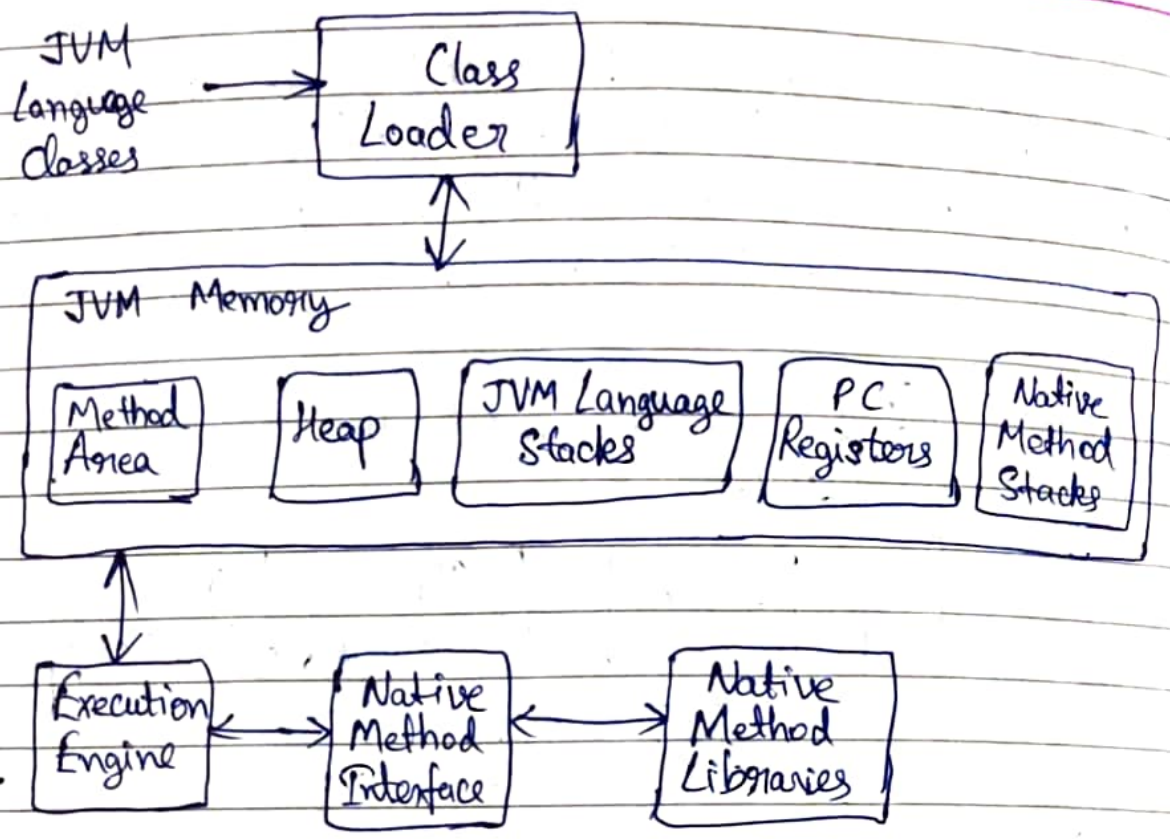
is mainly responsible for three activities :-

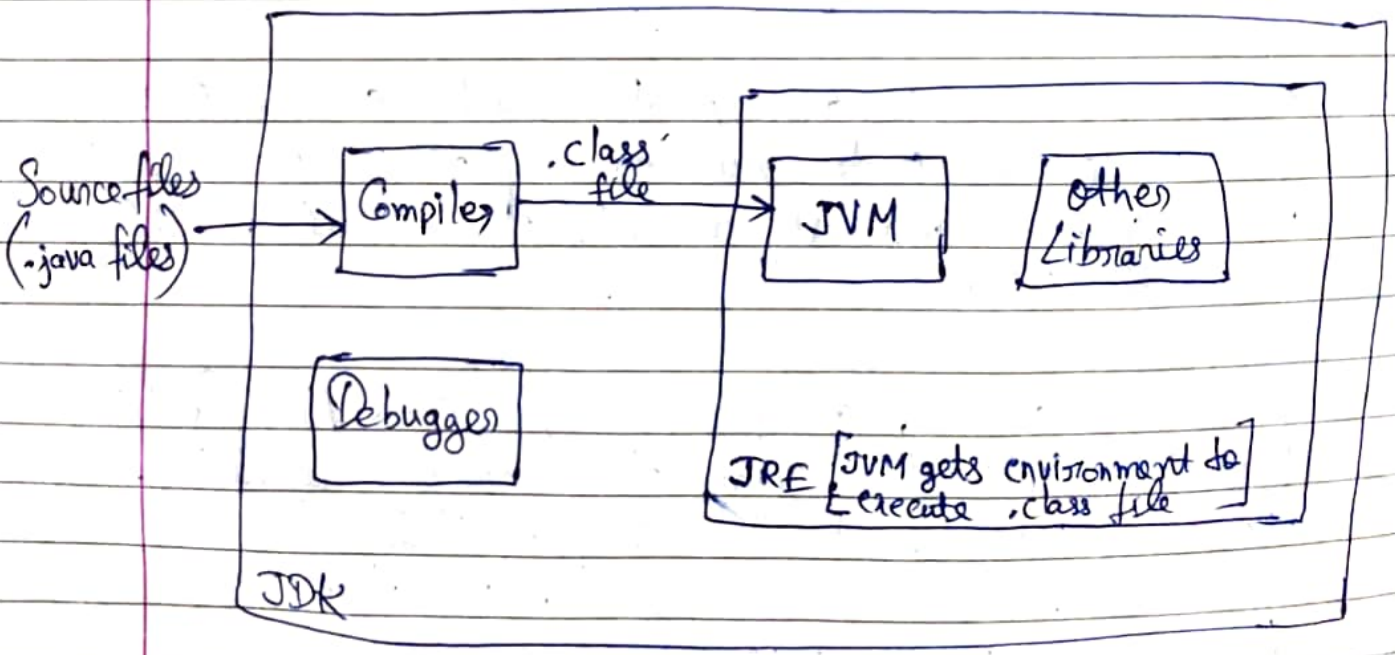
1. Loading
2. Linking
3. Initialization

1. **Loading** → The class loader reads the ".class" file, generate the corresponding binary data and save it in the method area. For each ".class" file, JVM stores the following information in the method area :-

   * fully qualified name of the loaded class and its immediate parent class.
   * whether the ".class" file is related to Class or Interface or Enum
   * Modifier, Variables and Method information etc.

After loading the ".class" file, JVM creates an object of type Class, to represent this file in the heap memory. Please note that this object is of type Class predefined in java.lang package. These Class object can be used by the programmer for getting class level information like the name of the class, parent name, methods and variable information etc. To get this object reference we can use getClass() method of Object class.





Java is platform-independent but JVM is platform dependent

DATE
PAGE

3. Initializing → In this phase, all static variables are assigned with their values defined in the code and static block (if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

In general, there are 3 class loaders :-

1. Bootstrap Class Loader

Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core Java API classes present in the "JAVA_HOME/jre/lib" directory.

bootstrap path = "JAVA_HOME/jre/lib"

2. Extension Class Loader

It is a child of the Bootstrap Class Loader. It loads the classes present in the extensions directories "JAVA_HOME/jre/lib/ext" (Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.

(4)

DATE
PAGE

2. Linking → Performs verification, preparation and (optionally) resolution.

Verification: It ensures the correctness of the .class file ie. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run time exception java.lang.VerifyError. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.

Preparation: JVM allocates memory for class variables and initializing the memory to default values.

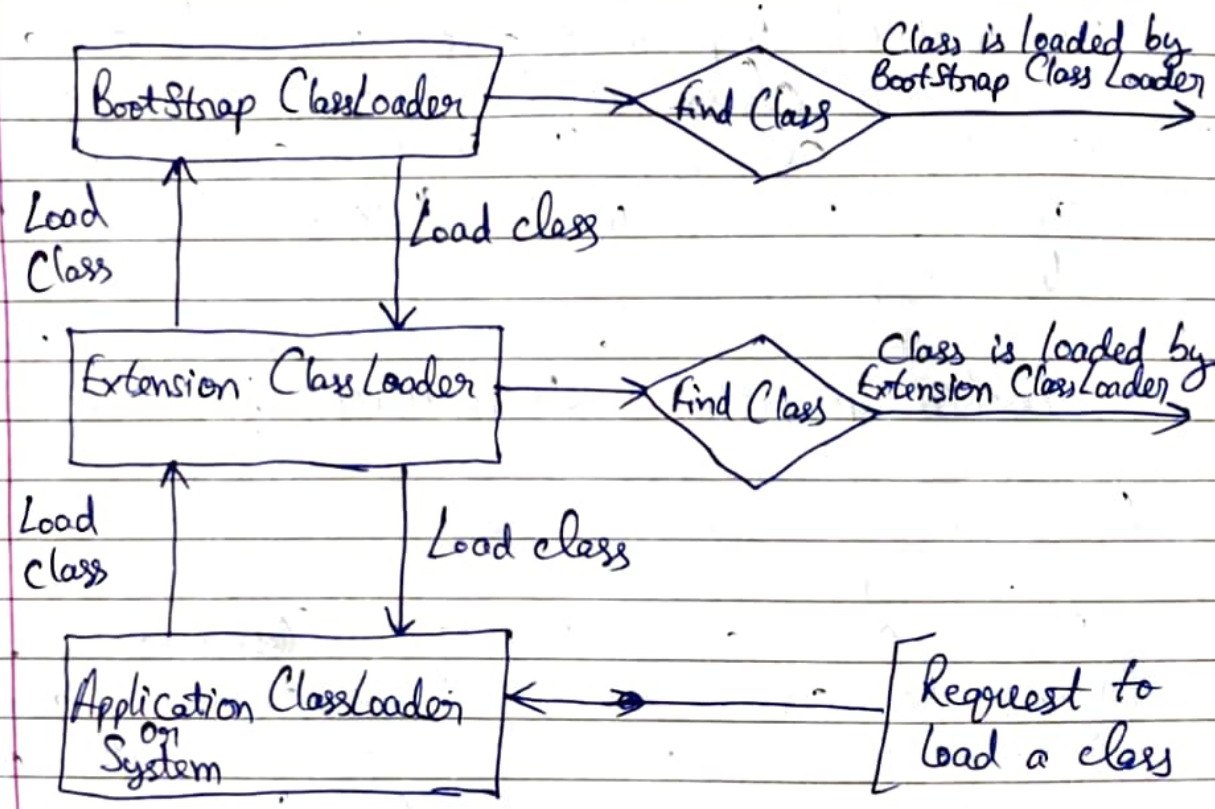
Resolution: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

DATE
PAGE

3. System/ Application Class Loader

It is a child class of extension class loader. It is responsible to load classes from the application classpath. It internally uses environment variable which mapped to java.class.path It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

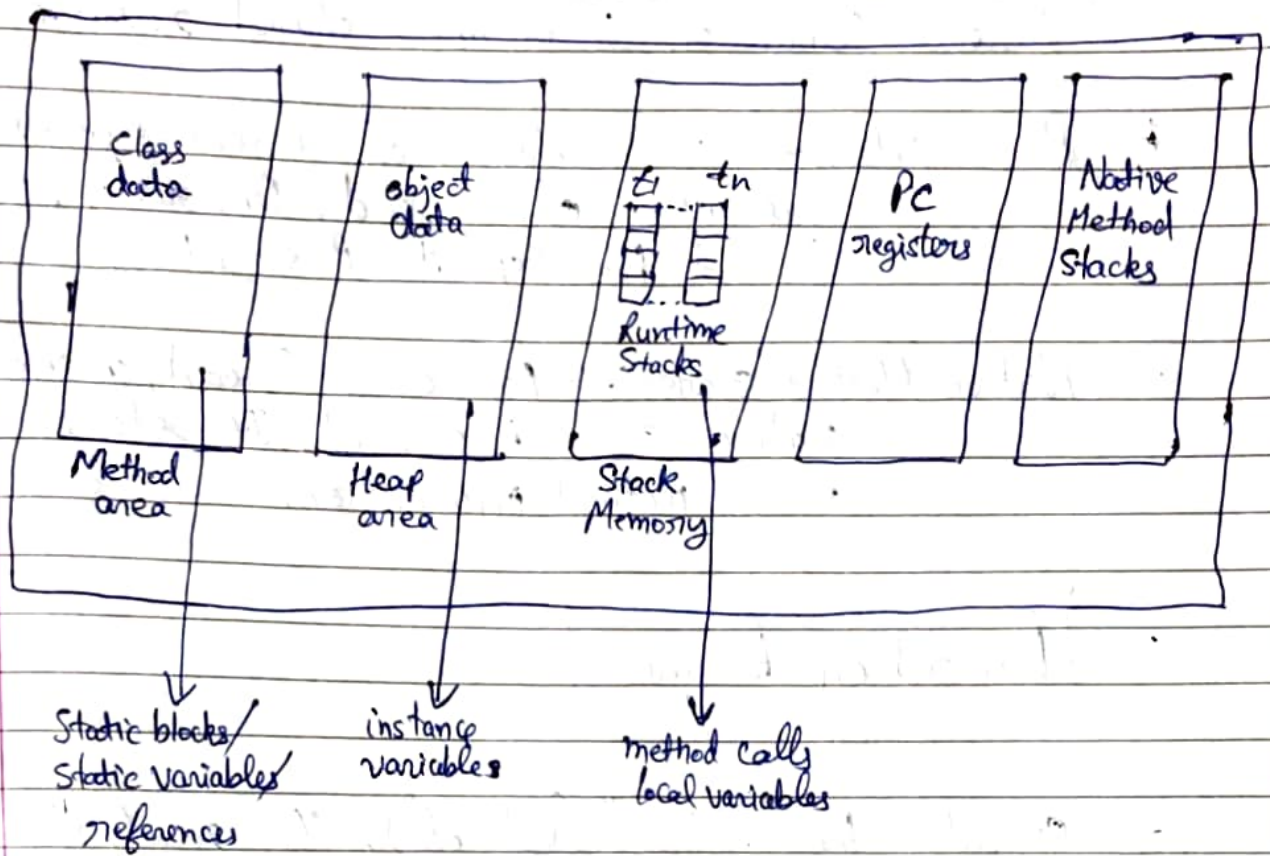
JVM follows delegation Hierarchy principle



At last, if the system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException

JVM Memory

DATE
PAGE



1. Method area → all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. Only one method area per JVM and it is a shared resource.

2. Heap area → Information of all objects is stored in the heap area. One heap area per JVM. It is a shared resource.

3. Stack area → For every thread, JVM creates one run time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame.

DATE
PAGE

Run time stack will be destroyed by JVM once a thread terminates. It is not a shared resource.

4. PC registers → store address of current execution instruction of a thread. Obviously, each thread has separate PC registers.

5. Native Method stacks → for every thread, a separate native stack is created. It stores native method information.

Execution Engine

→ executes the ".class" (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into 3 parts :-

- Interpreter → interprets the bytecode line by line and then executes.
  disadvantage : when one method is called multiple times, every time interpretation is required.

- Just-In-Time Compiler (JIT) → It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native

DATE
PAGE

code for that part so re-interpretation is not required, thus efficiency is improved.

- Garbage Collector → It destroys un-referenced objects.

Java Native Interface (JNI)

It is an interface that interacts with the native method libraries and provides the native libraries (C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which maybe specific to hardware.

Native Method Libraries

It is a collection of the Native Libraries (C, C++) which are required by the execution engine.

- Supported by Java (By default)

DATE
PAGE

① Data Types

⇒ Primitive Data Types

1) byte (8 bit) — -128 to +127 — -128, -127, ... 0, ... +126, +127

2) short (16 bit) — -32,768 to +32,767

3) int (32 bit) — -2^{31} to $+2^{31}-1$

4) long (64 bit) — -2^{63} to $+2^{63}-1$

Signed data types

absolute numbers and no decimal

5) float — Single precision — 32 bit IEEE 754 floating point — $3.40282347 \times 10^{38}$ to $1.40239846 \times 10^{-45}$

6) double — double precision — 64 bit IEEE 754 floating point — $1.79769313486231570 \times 10^{308}$ to $4.94065645841246544 \times 10^{-324}$

7) boolean — True / false — 1 bit of information

8) char — ['\u0000' (or 0) to '\uffff' (or 65,535)] — 16 bit Unicode character

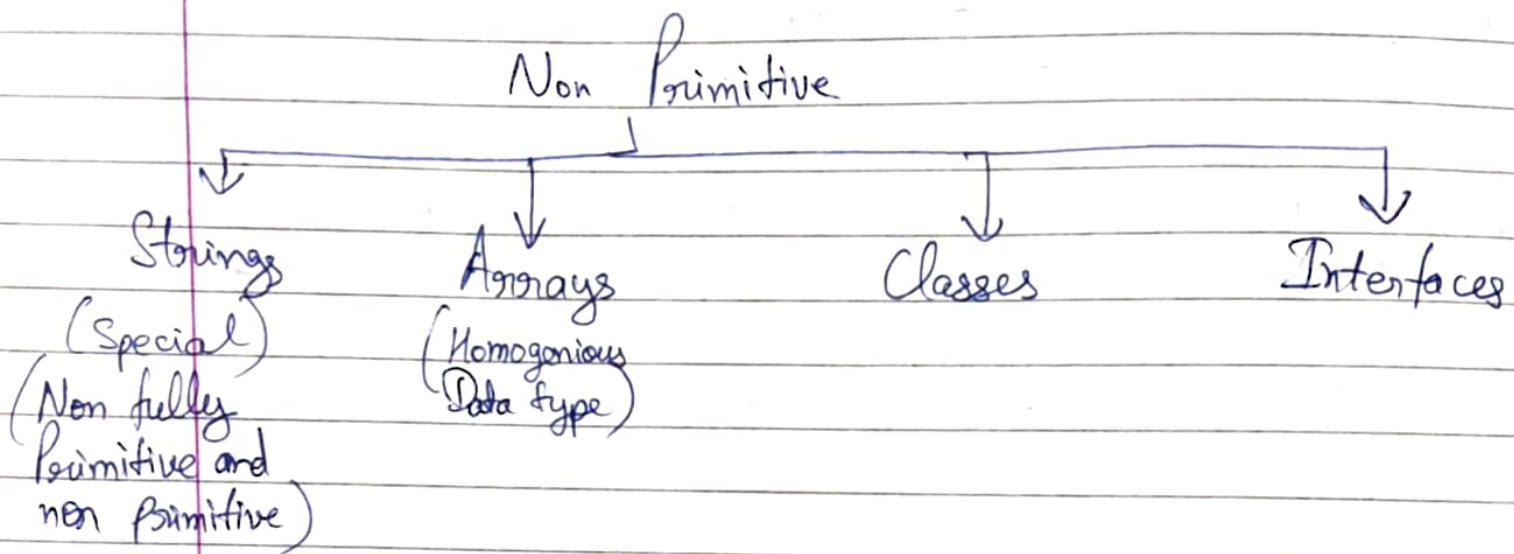
character → unicode → binary

binary → unicode → character

• Custom in nature (we can develop)

DATE
PAGE

⇒ Non Primitive Data types



Arrays

1. int arr[] = new int [2];
   arr[0] = 1;
   arr[1] = 6;

2. int arr[] = {1, 6};

3. int arr[] = new int[] {1, 6};

⇒ Tokens in Java

Tokens are reserved expressions, symbols, words which have a pre defined meaning in Java. Cannot Overwrite.

1) Keywords

2) Identifiers
   ↳ any variable names you declare in your program.
   cannot: start with number, contain space, have +, -, &

3) Constants
   ↳ final variables

4) Special Symbols
   [] () {}, ; * =

5) Operators
   ↳ arithmatic + - * /
   comparison
   Logical
   bitwise etc

DATE
PAGE

⇒ Data Type Conversion

→ Implicit Data Type Conversion / Casting

* Smaller range variable into larger range variab

```
int a = 100;

long b = a;

float c = b;
```

→ Explicit Data Type Conversion / Casting

* larger range variable to smaller range varia
* Quality sacrificed.

```
double a = 50.50;

float f = (float) a;

long l = (long) f;

int c = (int) l;
```

⇒ Hello World program in Java.

⇒ Operators in Java

1. Assignment Operators
   =

2. Arithmetic Operators
   + - * / %
   (% modulus)

3. Unary Operators
   + - ++ -- !

4. Equality, Relational Operators
   == equal to
   != not equal to
   > greater than
   >= greater than or equal to
   < less than
   <= less than or equal to

5. Conditional Operators
   && Conditional-AND
   || Conditional-OR
   ? : ternary operator
   if-then-else
   ? :

} Comparison Operators

&& → short circuit operator
& → standard operator

|| → short circuit operator
| → Standard operator

⇒ Control Flow statements

* if - then statement
* if - then - else statement
* Nested if statement
* Switch Statement

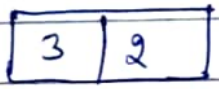
⇒ 2D Array in Java

```
int arr[][] = { {2,7,9}, {3,6,1}, {7,4,2} }
```

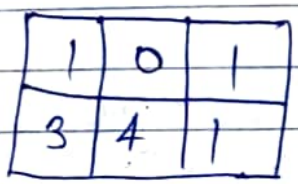
	col 0	col 1	col 2
row 0	2	7	9
row 1	3	6	1
row 2	7	4	2

⇒ 3D Array in Java

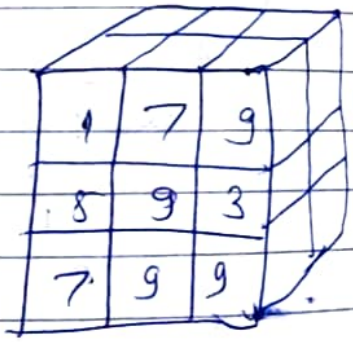
1D



2D



3D




```
int[][][] arr
  = { { {1,2,10}, {3,4,11} }, { {5,6,12}, {7,8,13} } }
```

DATE
PAGE

⇒ Loops

* Do while
* while
* for loop
* enhanced for loop
* nested for loop

⇒ Classes in Java

* A class is a blueprint of an object.
* An object is an instance of a class.
* A class is a logical entity with attributes (i.e. properties) and behaviour (i.e. methods)

⇒ Constructors in Java

* Constructors pass parameters
* When we dont provide a constructor to a class then java will by default provide a constructor which will be a no parameter constructor.
* Default constructor will only work if we have not provided any constructor.

⇒ OOPs

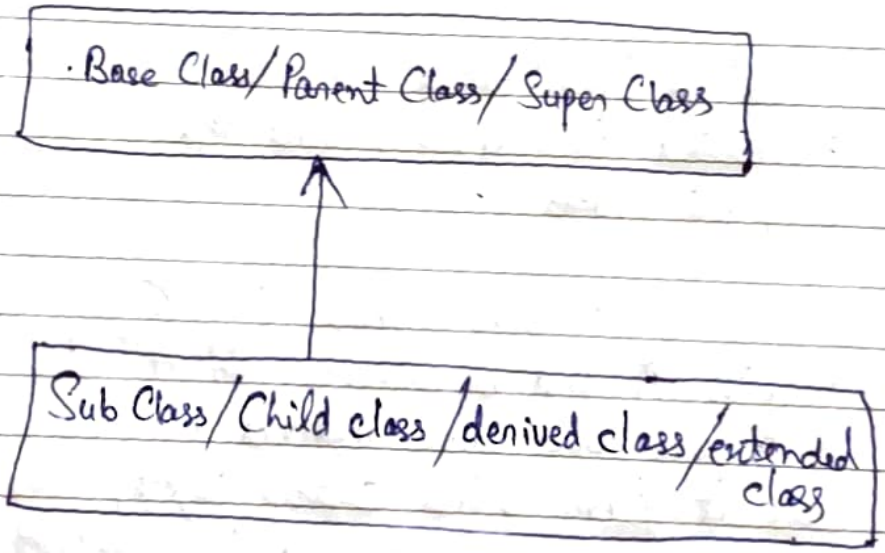
"Object oriented Programming." is a programming paradigm based on the concept of classes and objects.

Four main principles of OOP are:-

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism.

1.) Inheritance

In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.



is – a relationship

⇒ OOPs

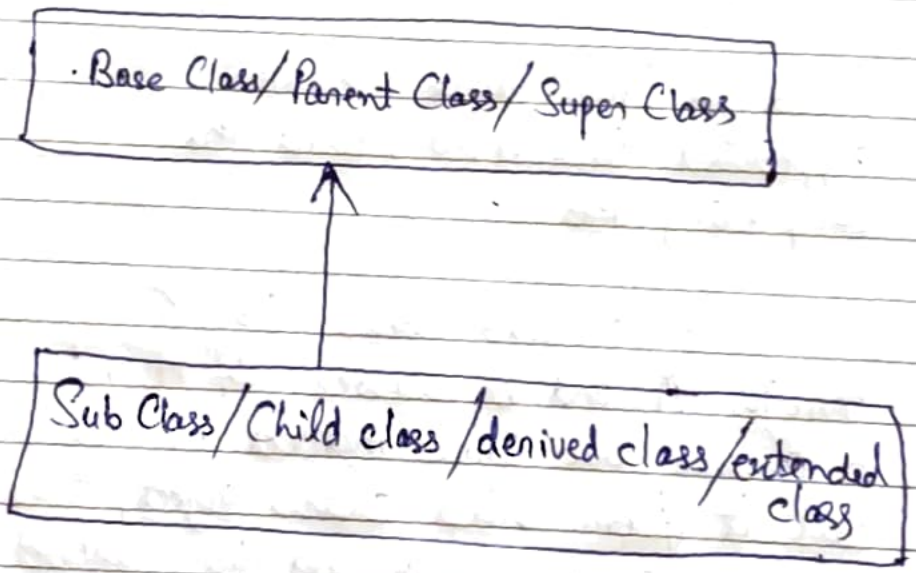
"Object oriented Programming." is a programming paradigm based on the concept of classes and objects.

Four main principles of OOP are :-

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism.

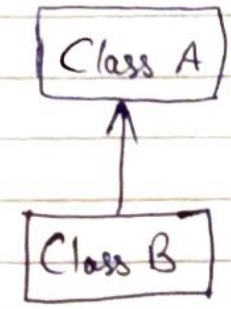
1.) Inheritance

In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.



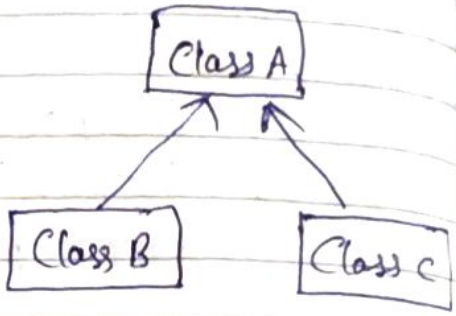
is – a relationship

Except Object class, which has no superclass, every class has one and only one direct superclass. In the absence of any other explicit superclass, every class is implicitly a subclass of Object.



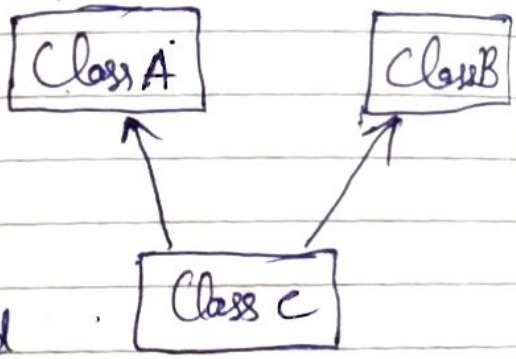
Single

Multilevel



Hierarchical

Multiple Inheritance is not supported in Java to reduce the complexity.



Consider a scenario where Class C inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

⇒ Encapsulation

* Wrapping up of data.
* It is a protective shield that prevents the data from being accessed by the code outside the shield.
* In encapsulation, the variables or data of a class is hidden from any other class & can be accessed only through any member function of its own class in which it is declared.
* Declare all the variables in the class as private and write public methods in the class to set and get the values of variables.

Advantages of ~~data~~ Encapsulation :-

1. Data Hiding
2. Increased Flexibility
3. Reusability
4. Testing code is easy.

Super

⇒ Uses of super keyword :-

1. To call methods of the superclass and even called overridden methods in the subclass

2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.

3. To explicitly call super class no-arg (default) or parameterized constructor from the subclass constructor.

⇒ Static Keyword in Java

* to create fields and methods that belongs to the class rather than to an instance of the class.

* **Static variable**
  - ↳ become the property of the class and not of object

* **Static methods**
  - ↳ methods belong to the class and not to the object
  - ↳ a common use of static methods is to access static methods
  - ↳ static methods can only access static variables.
  - ↳ static methods can not be overridden.
  - ↳ instance methods can access static variable.

* **Constants**
  - ↳ the static modifier, in combination with the final modifier, is also used to define constants.

eg.→ `static final double PI = 3.14159265358979 3;`

* **Static blocks**
  - ↳ can only access static variables
  - ↳ block of logic which you want to execute only once when class is loaded in run time.

```
static {

}
```

⇒ Abstraction

* Hide some detail of an object.

abstract can be used with a class or a method

* abstract class

→ An abstract class is a class that is declared abstract.

→ It may or may not include abstract methods.

→ It cannot be instantiated but they can be subclass.

→ abstract class can also have concrete methods.

→ If a child class is not ~~implimenting all~~ providing implementation of all abstract methods of parent abstract class then make the child class as abstract class.

* abstract methods

→ abstract method is a method that is declared without an implementation.

→ abstract method can exist only in an abstract class.

⇒ Interface in Java

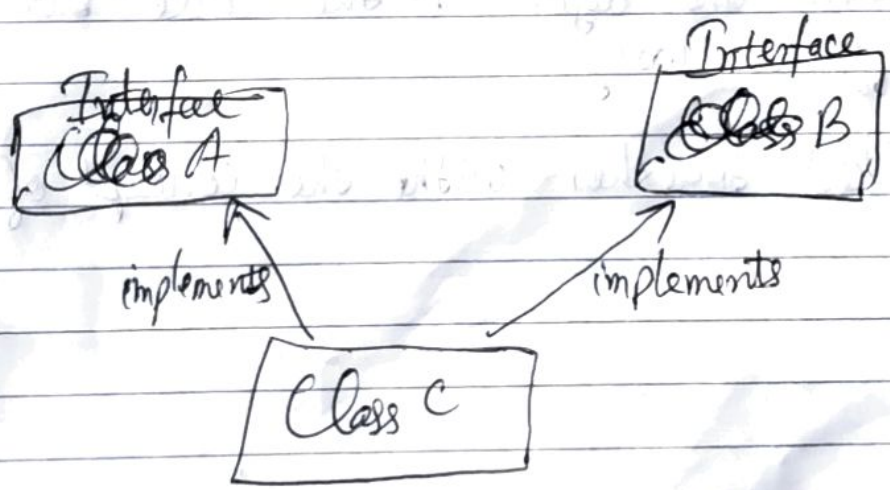
Interface define the specification of how a class would act.

Interface define the contract of how a code, programming logic will act.

- constants
- method signature
- default methods (new in Java 8) ⇒ to allow developers to add new methods to the interfaces without affecting the classes that implements the interface.
- static methods (new in Java 8)
- nested types

* NO Concrete method.
* Methods are public & abstract by default.
* public static final variables.
* only declare method

Interfaces are more restrictive.



Multiple Inheritance with interface is possible. One class can implement more than 1 interface.

⇒ Polymorphism

↳ A particular object/entity can take multiple forms.

entity means class, object, method.

* Method Overloading

↳ Name of method should remain same.
return type of methods can change
return type of arguments can change
Number of arguments can change.

Java choose the method based on number of arguments and return type of argeiments.

* Method Overriding

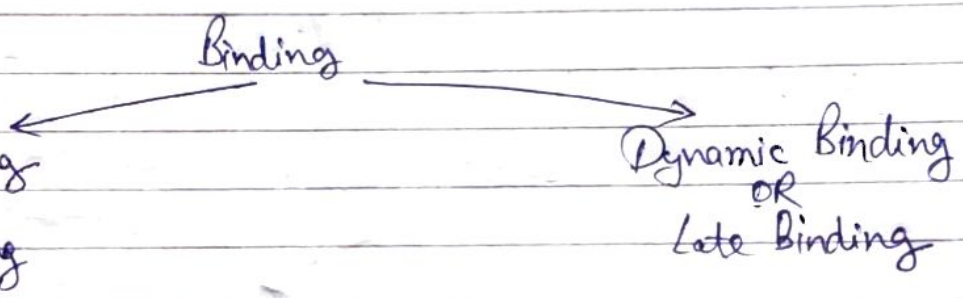
↳ Same signature of the method being used again and again in the code but in a different way

↳ Used together with the concept of inheritance.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile time error
Subclass Static Method	Generates a compile time error	Hides

Association of method definition to the method call is known as binding

Two types of binding



1. Type of object is determined at compile time.	1. Type of object is determined at run time.
2. Private, final and static methods and variables uses static binding	2. Virtual methods use dynamic binding
3. Uses type of the class and field to resolve.	3. Uses object to resolve binding.
4. Overloading is the example of static binding	4. Method overriding is the example of Dynamic binding.
5. Compile time polymorphism	5. Run time polymorphism

⇒ Final Keyword in Java

* Java final Variable

If we make any variable as final, we cannot change the value of final variable. It will be constant.

```
final int speedlimit = 140;
```

* final method

A final method cannot be overridden.

* final class

If we make any class as final, we cannot extend it.

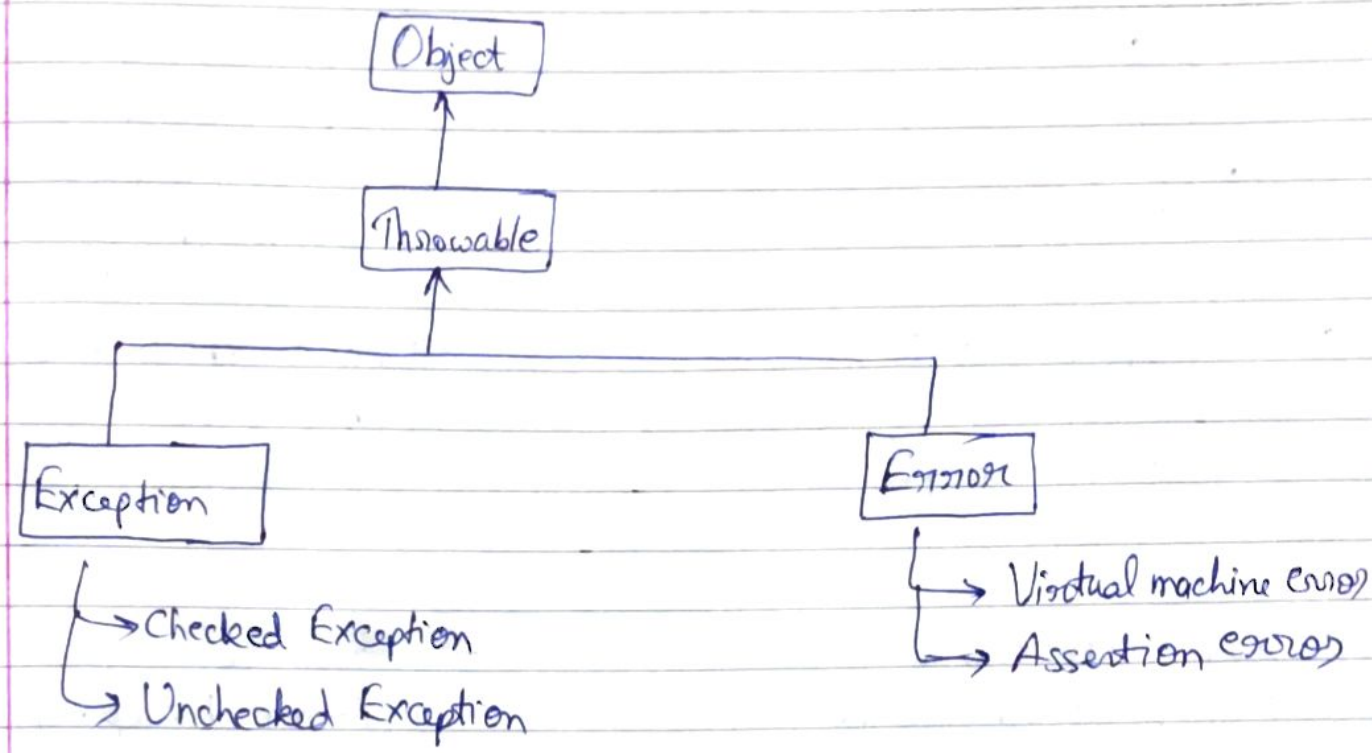
* blank or uninitialized final variable

↳ not initialized at the time of declaration.
↳ It is useful when we want to create a variable that is initialized at the time of creating object and once initialized may not be changed. example → PAN card of an emplogee.
↳ It can be initialized only in constructor.

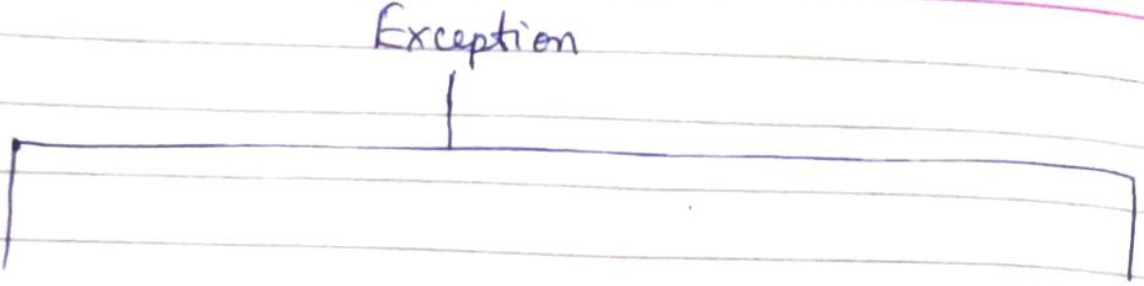
* Static blank final variable

↳ not initialized at the time of declaration.
↳ It can be initialized only in static block.

⇒ Exceptions in Java



Error	Exception
1. Impossible to recover from error	1. Possible to recover from exception.
2. Error are of unchecked type	2. Exception can be checked or unchecked.
3. Happen at runtime	3. Can happen at compile time and runtime.
4. Caused by the environment on which application is running	4. Caused by application.



Checked

→ An exception that is checked by the compiler at compilation time

→ These exceptions cannot simply be ignored, the programmer should handle these exceptions

eg. FileNotFoundException
SQLException
IOException

Unchecked (RuntimeException)

→ An exception that occurs at the time of execution

→ Also called as RuntimeException

→ RuntimeException are ignored at the time of compilation

eg. ArithmeticException
IndexOutOfBoundsException
NullPointerException

* Custom Exception

```
class MyException extends Exception
    {

    }
```

Exception Handling

*
```
try {

} Catch (ExceptionType name) {

} Catch (ExceptionType name) {

    }
```

Catch block contains code that is executed if and when the exception handler is invoked.

finally
{
}

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So its a perfect place to perform cleanup.

We can write the finally block without the catch block ~~also~~ as well.

* throws → used to declare an exception
  → always used with method signature

How JVM handles exception
↳ inside a method if an exception has occured
↳ methods creates an object known as exception object and hands it off to the runtime system
↳ this exception object contains name and description of the exception and also current state of the program where exception occured

↳ then using try catch finally ~~methods~~ blocks these exceptions can be handled

Custom Exception

We can create our own exceptions in Java. It is known as Custom exception / User defined exception.

```
public class CustomException extends Exception
  {
     public CustomException(String errormessage)
        {
           super(errormessage);
        }
  }

public static void main(String[] args) throws CustomExceptio
  { int x = 30;
    if (x<40) {
     throw new CustomException("invalid");
    }
  }
```

Reasons to use Custom Exception :-

1. To catch and provide specific treatment to a subset of existing Java exceptions.
2. Business logic exceptions.

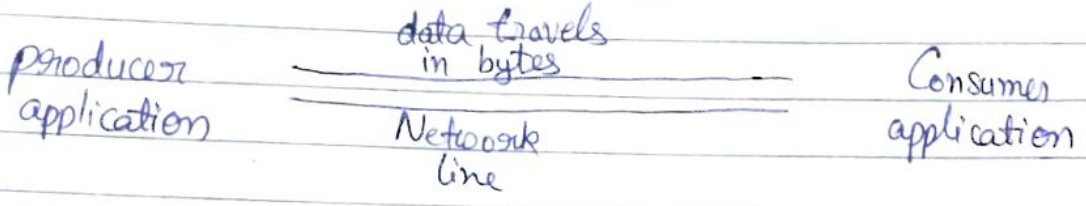
Difference between throw and throws

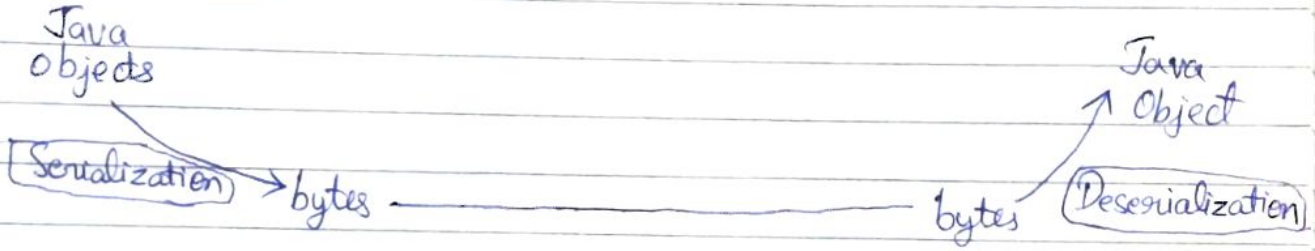
	throw	throws
1.	Used to explicitly throw an exception	1. Used to declare an exception
2.	Checked exceptions cannot be propagated without using throws keyword.	2. Checked exception can be propagated
3.	followed by an instance	3. followed by a class
4.	Used within a method	4. Used with a method signature.
5.	Cannot throw multiple exception	5. Can declare multiple exceptions.

⇒ Serialization

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object.

When you are building enterprise grid applications in different organizations you will often have a need to send an object from one application to other application which is deployed in some other machine.





java.io.Serializable
java.io.Externalizable

When you serialize the object, the class type information is lost.

transient → If you do not want any variable to be serialized or deserialized.

Serialization	Deserialization
FileOutputStream	FileInputStream
ObjectOutputStream	ObjectInputStream
↘	↙
writeObject	readObject

⇒ Concurrency and Threads in Java

* Processes and Threads

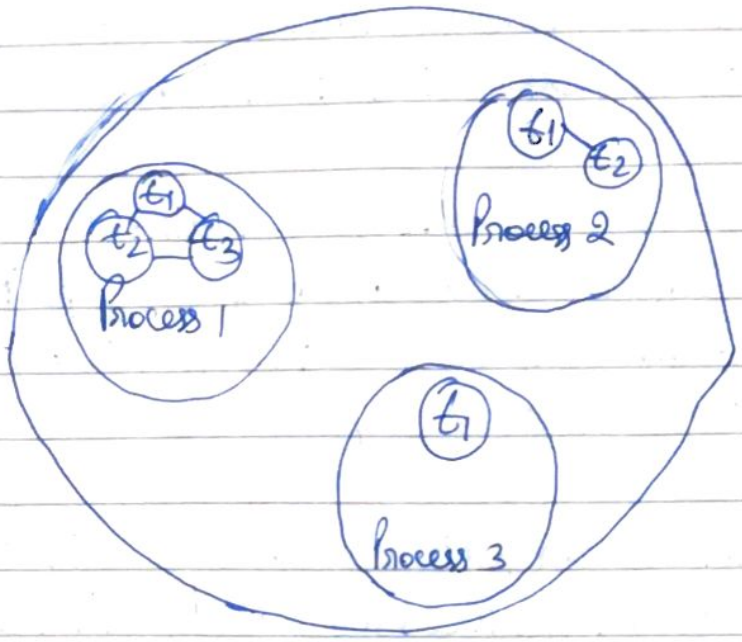
In concurrent programming, there are two basic units of execution: processes and threads

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources: in particular, each process has its own memory space.

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient but potentially problematic, communication.

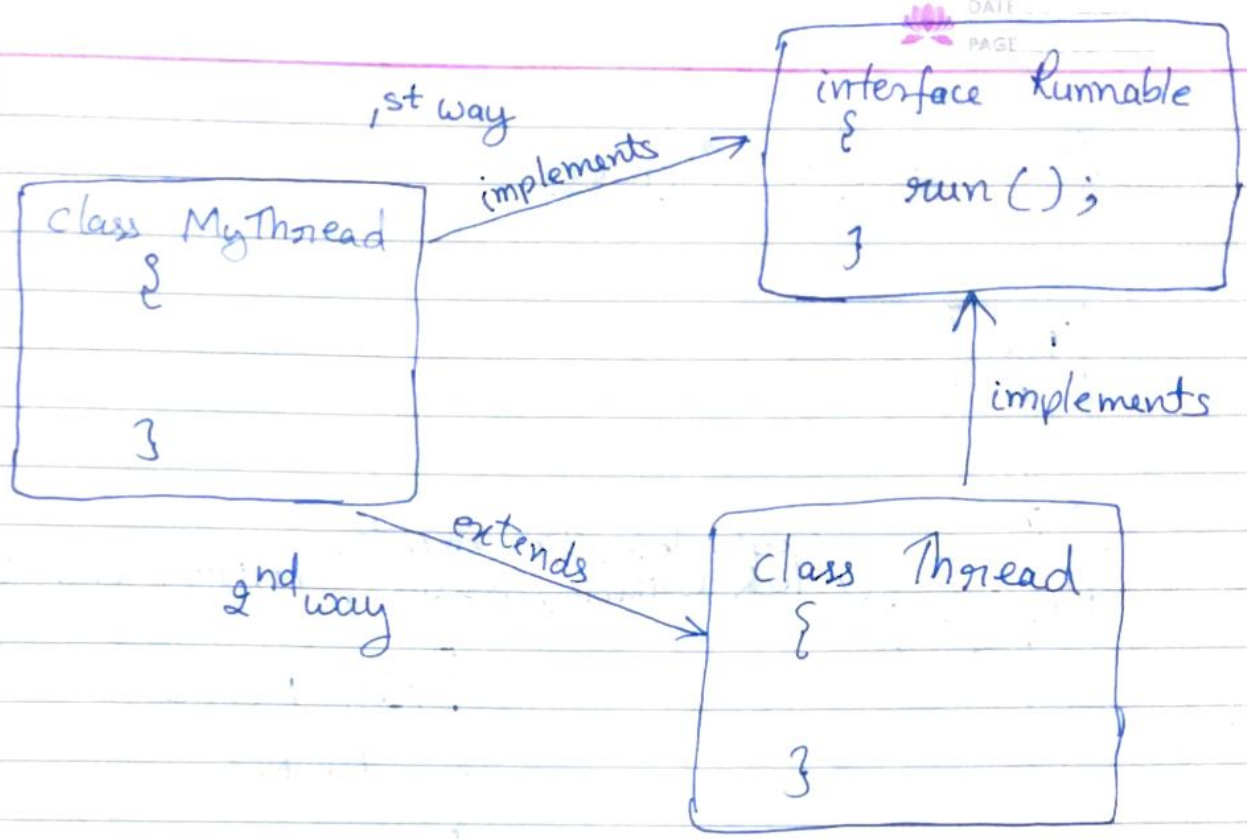
Multithreaded execution is an essential feature of the Java ~~application~~ platform. Every application has at least one thread – or several.



→ A Java program is a process

→ Creating Threads in Java

* Using Runnable Interface
* Using Thread class.



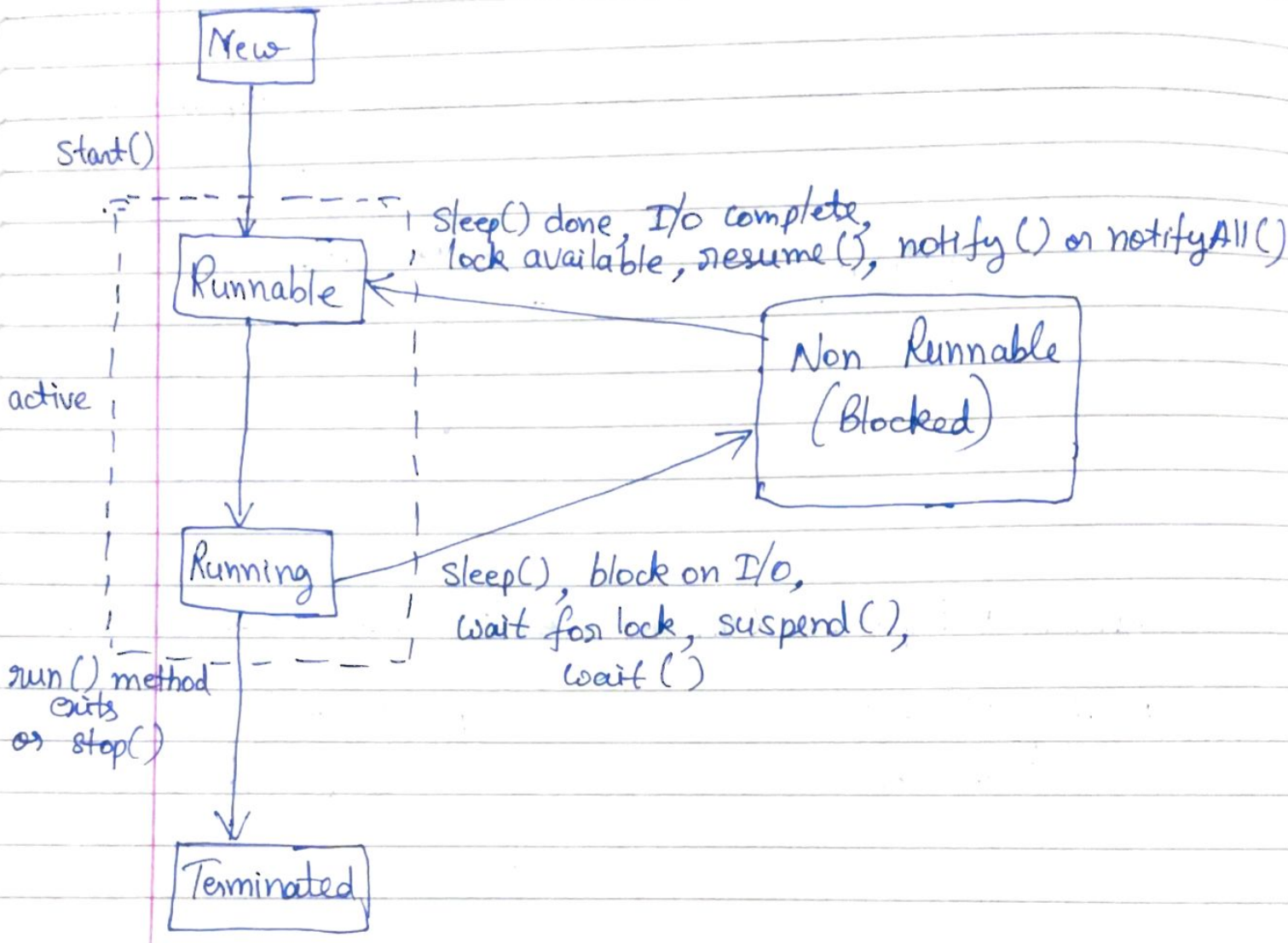

```
class MyThread implements Runnable
{
    public void run()
    {
        // task
    }
}

public static void main(String args[])
{
    MyThread t1 = new MyThread();
    Thread thread = new Thread(t1);
    thread.start();
}
```

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

public static void main(String args[])
{
    MyThread t = new MyThread();
    t.start();
}
```

Life cycle of Thread



⇒ Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.

When a thread try to work on a particular object, it basically takes a lock on that particular object. It means that the particular object becomes inaccessible to other threads in the system at that particular time.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when its done with them.

As long as a thread owns an intrinsic lock, no other thread can acquire the same lock.

If your code is executing in a multi-threaded environment, you need synchronization for objects, which are shared among multiple threads, to avoid any corruption of state or any kind of unexpected behaviour. Synchronization in Java will only be needed if a shared object is mutable.

DATE
PAGE

⇒ # Daemon Thread

Daemon thread is a low priority thread that runs in background to perform tasks such as Garbage Collection.

Daemon thread in java is a service provider thread that provide services to user thread.

Garbage collector is best example of Daemon thre

JVM terminates itself when all user threads (non-daemon) finish their execution, JVM does not care whether Daemon thread is running or not.