***How is a C++ program executed?***

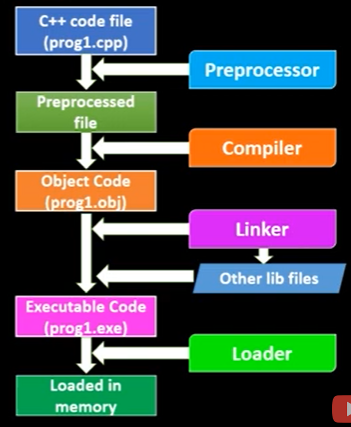
**Writing Code:** You write your C++ program using a text editor or an integrated development environment (IDE). (VS code, CodeBlocks, NetBeans, Eclipse, Pycharm, Android Studio)

**Compilation:** You use a C++ compiler (such as GCC, Clang, or Microsoft Visual C++) to compile the source code. This process creates object files (files with a ".o" or ".obj" extension) for each source file.

1. Lexical Analysis (Scanning):
   * The first stage involves breaking the source code into tokens. This is done by a component called the lexer or scanner.
   * The lexer recognizes keywords, identifiers, literals, and other syntactic elements in the source code.
2. Syntax Analysis (Parsing):
   * The parser takes the tokens generated by the lexer and organizes them into a hierarchical structure called the abstract syntax tree (AST).
   * The AST represents the grammatical structure of the source code and is used to ensure that the code adheres to the rules of the programming language.
3. Semantic Analysis:
   * The semantic analyzer checks the meaning of the source code. It verifies that the code is semantically correct and adheres to the language's specifications.
   * It performs tasks such as type checking, scope resolution, and checking for semantic errors.
   * Incorrect Variable Assignment, Logical Error in Condition, Array Index Out of Bounds, Infinite Loop.

**Linking:** If your program consists of multiple source files or uses external libraries, the linker combines the object files into a single executable file.

**Execution:** The user runs the executable file( files with .exe extension) , and the operating system loads it into memory. The CPU executes the machine code instructions generated by the compiler, and the program performs its intended tasks.



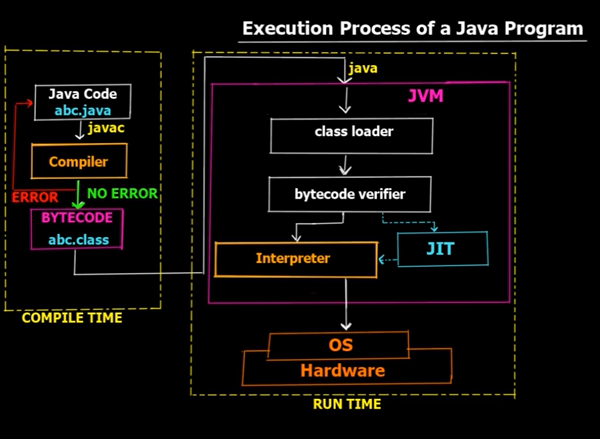
***How is a java program executed?***

**Writing Code:** Developers write Java code using a text editor or an integrated development environment (IDE).

**Compilation:** The Java compiler (**javac**) translates the human-readable Java source code into an intermediate form called bytecode. Bytecode is a low-level representation of the code that is not specific to any particular hardware or operating system.

**Java Virtual Machine (JVM):** Java bytecode is not directly executed by the computer's hardware. Instead, it is executed by the Java Virtual Machine (JVM) in Java Runtime Environment(JRE). To execute Java code, the Java bytecode is interpreted or, more commonly, Just-In-Time (JIT) compiled by the Java Virtual Machine (JVM). The JVM is platform-specific, but the bytecode is platform-independent. This allows Java programs to be written once and run anywhere (the principle of "write once, run anywhere" or WORA). JVM provides a runtime environment for Java programs.

**Execution:** The JVM loads the bytecode and dynamically translates it into machine code that is specific to the host system. This process happens at runtime. The CPU then executes the machine code, and the Java program runs as intended.



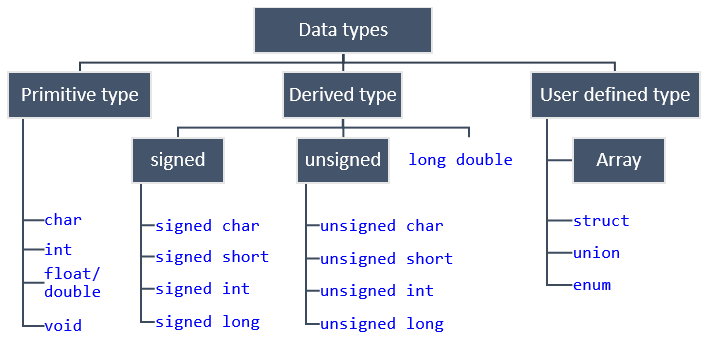
***What is JVM?***

The Java Virtual Machine (JVM) is a runtime environment that executes Java bytecode. It ensures platform independence by interpreting or compiling bytecode into native machine code at runtime. The JVM manages memory, provides security features, dynamically loads classes, and facilitates the execution of Java programs. It acts as an abstraction layer, allowing Java applications to run on diverse platforms without modification.

**JRE (Java Runtime Environment):** The JRE is a set of software tools that provides the runtime environment for executing Java applications and applets. It includes the Java Virtual Machine (JVM), class libraries, and other necessary files to run Java applications.

**JDK (Java Development Kit):** The JDK is a software development kit used for developing Java applications and applets. It includes the JRE along with development tools such as a compiler (javac), debugger (jdb), and other utilities needed for Java development.

Primitive data type. C language supports four primitive types – char , int , float , void



In C, an **enum** (short for enumeration) is a user-defined data type that consists of a set of named integer constants.

enum Color {

RED,GREEN,BLUE

};

int main() {

enum Color myColor;

myColor = RED;

switch (myColor) {

case RED:

printf("The color is red.\n");

break;}

The **malloc** function takes a single argument, **size**, which represents the number of bytes to allocate. It returns a pointer to the beginning of the allocated memory block. It's important to note that the memory returned by **malloc** is uninitialized, meaning it may contain arbitrary values. If you need to initialize the allocated memory, you can use functions like **calloc** (which initializes the memory to zero) or manually set the values. After using the allocated memory, it's essential to free it using the **free** function to avoid memory leaks. Additionally, C++ provides features like **new** and **delete** for similar purposes in an object-oriented context.

***common use of #include <stdlib.h> library in c*** : malloc(), Contiguous Allocation calloc(), realloc(),free(), atoi(), atoll(), atol(), atof(), exit(), rand(), srand(), abs().

**ctype.h : isalpha, isdigit, islower, isupper, toupper, tolower**

***what are the main difference between struct( which is in C) and class( which is in C++)***

**Default Member Access:** In a C struct, all members have public access by default. In a C++ class, all members have private access by default.

**Member Functions:** C structs do not support member functions. Functions that operate on struct data are defined separately. C++ classes can have member functions, which are functions that operate on the class's data.

**Constructor and Destructor:** C structs do not have constructors or destructors. C++ classes can have constructors

**Sizeof operator will generate 1 for an empty string.** This is because a C string is terminated by a null character ('\0'), which takes up one byte of memory.

**Sizeof operator will generate 1 for an empty structure.** According to the C standard (C11 standard, section 6.7.2.1), an object in C must have a size greater than zero. Point to be noted: in local machine I have run an empty struct and the size prints 0;

**Pointers:** A pointer is a variable that stores the memory address of another variable.

int \*ptr;

    \*ptr = 5;// stores the address where 5 is located in the memory.

std::cout << &ptr << '\n';// 0x61fe10 memory address of the pointer

std::cout << ptr << '\n';// 0xfd15f0 memory address of the value 5

    std::cout << \*ptr << '\n';// 5 deferencing- gets the value in the address

saved in the pointer

**Dereferencing:** **\*ptr** dereferences the pointer, providing access to the value stored at the memory address it points to.

**ptr++;** moves the pointer to the next memory location.

**Reference:** In C++, a reference is an alias or an alternative name for an existing variable. References provide a way to access the same memory location as the original variable using a different name.

void modifyValue(int &ref) {

ref = 123;

}

int main() {

int value = 42;

modifyValue(value);

std::cout << "Modified Value: " << value << std::endl;

return 0;

}

when you pass an array to a function in C++, it is passed by pointer. The following both are same.

void printArray(int \*arr)

void printArray(int arr[])

Classes are user-defined data types in C++ that encapsulate data (attributes) and functions (methods) operating on the data.

Encapsulation is the bundling of data and methods that operate on that data into a single unit (class).

**Key Differences between abstraction and encapsulation:**

* **Focus:**
  + Abstraction focuses on creating a high-level, conceptual view of objects, emphasizing what objects do.
  + Encapsulation focuses on bundling related data and methods within a class, controlling access to the internal state and promoting data hiding.
* **Implementation:**
  + Abstraction is more about defining abstract classes, interfaces, or abstract data types.
  + Encapsulation is about implementing classes and controlling access to their members.
* **Level of Detail:**
  + Abstraction hides unnecessary details to provide a clear and conceptual view of objects.
  + Encapsulation hides the internal implementation details of a class to ensure a controlled and secure interface.

A virtual function is a function in a base class (or superclass) that is declared with the **virtual** keyword and is meant to be overridden by derived classes (or subclasses).

#include <iostream>

using namespace std;

class Base {

public:

    virtual void myFunction() {

        cout << "Base class implementation\n";

    }

};

class Derived : public Base {

public:

    void myFunction() override {

        cout<<"Derived class implementation\n";

    }

};

int main() {

    Base\* basePtr;

    Derived derivedObj;

    basePtr = &derivedObj;

    // Calls the overridden function in the derived class

    basePtr->myFunction();

    return 0;

}

//Output: derived class implementation

If the virtual keyword is not used then,

//Output: Base class implementation

***Friend Function:*** A friend function of a class is defined outside that class scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend function appear in the class definition, friends are not member functions.

#include <iostream>

using namespace std;

class Distance{

    int meters;

    public:

    Distance()

    {

        meters = 0;

    }

    void printmeter()

    {

        cout << "The distance is " << meters << '\n';

    }

    friend void add(Distance &d);

};

void add(Distance &d)

{

    d.meters += 5;

}

int main()

{

    Distance d1;

    d1.printmeter();

    add(d1);

    d1.printmeter();

}

***C++ files and streams:***

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    ofstream os;

    os.open("text.txt");

    string line;

    while (getline(cin, line))

        os << line<<'\n';

    os.close();

    ifstream is;

    is.open("text.txt");

    while (getline(is, line))

        cout << line << '\n';

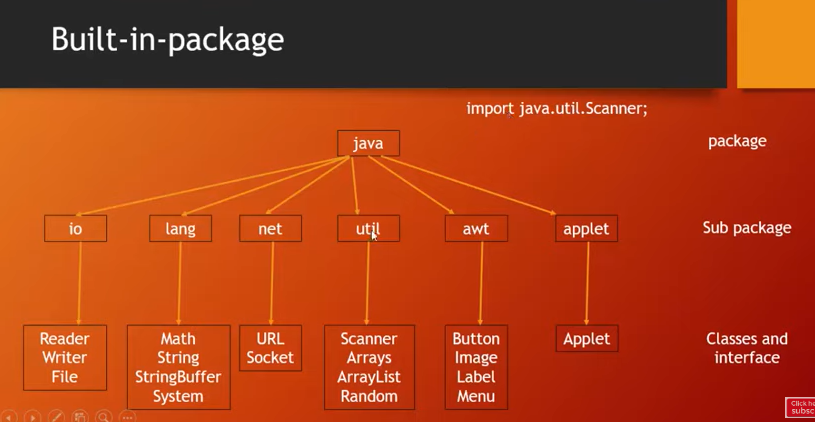
    os.close();}

***Iterators:*** vectors occupy adjacent locations in memory whereas maps, sets use memories which are in different locations. v.begin() is an iterator which points to the first element of the vector.

(It+1) iterator moves to the next location. Which is not ok for containers like map, set as they allocate memory form different locations in the memory.

It++ iterator moves to the next iterator. Ok for every container.

***Packages:***



**Context switching** is a process that occurs in multitasking operating systems to switch the focus of the central processing unit (CPU) from one task or process to another. Context switching involves saving the state of the currently executing task (context) so that it can be later restored when the task is resumed, and then loading the saved state of the next task to be executed.

"overhead" refers to the additional resources, time, or processing power that is required by a system or a process beyond the minimum necessary to perform a specific task.

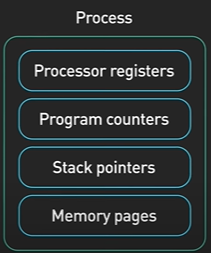
***Process:***

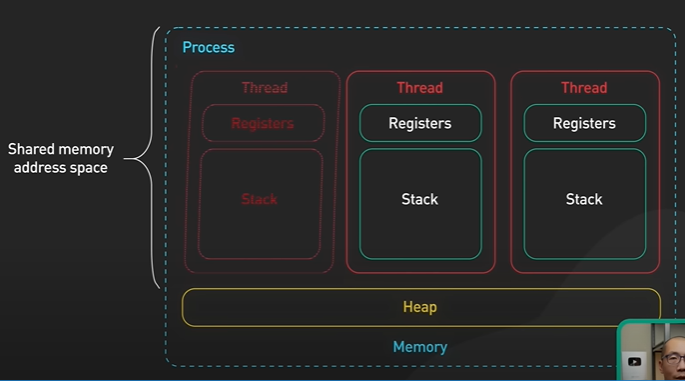
A process is an independent program that runs in its own memory space and has its own resources. Processes are isolated from each other. Processes have more overhead compared to threads because they are more independent and require separate memory space and resources.

***Thread:***

A thread is a lightweight process that exists within a process and shares the same resources, which means they can directly access each other's data. Threads have less overhead compared to processes since they share resources. Context switching between threads is typically faster.

Threads are created and terminated within a process. They share the life cycle of the process. If one thread in a process crashes, it can potentially affect the entire process.

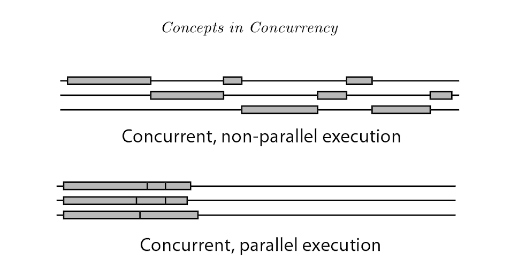




Program counter stores the address of the next instruction to be executed.

***Concurrency:*** At any given moment, the CPU is working on one task while others may be in various states of progress or waiting. The goal of concurrency is to improve the efficiency of resource utilization and to design systems that can handle multiple tasks without necessarily requiring true parallelism.

***Parallelism:*** In a parallel system, multiple processors or cores work simultaneously on different parts of the same task, actively executing tasks at the same time. The goal of parallelism is to enhance performance by dividing a task into subtasks that can be executed concurrently. The aim is to achieve a speedup in overall execution time.



***Multithreading:*** Ways To Create A Thread In Java

1. By extending the thread class
2. By implementing a Runnable interface

class MyThread1 extends Thread{

    public void run(){

        int i =0;

        while(i<40000){

            System.out.println("My Cooking Thread is Running");

            System.out.println("I am happy!");

            i++;

        }

    }

}

class MyThread2 extends Thread{

    public void run(){

        int i =0;

        while(i<40000){

            System.out.println("Thread 2 for Chatting with her");

            System.out.println("I am sad!");

            i++;

        }

    }

}

public class cwh\_70 {

    public static void main(String[] args) {

    MyThread1 t1 = new MyThread1();

    MyThread2 t2 = new MyThread2();

    t1.start(); //run() function is inside the start() function which

    t2.start(); automatically calls the run() function.

    }

}

class MyThreadRunnable1 implements Runnable{

    public void run(){

        System.out.println("I am a thread 1 not a threat 1");

    }

}

class MyThreadRunnable2 implements Runnable{

    public void run(){

        System.out.println("I am a thread 2 not a threat 2");

            }

}

public class cwh\_71\_runnable {

    public static void main(String[] args) {

        MyThreadRunnable1 bullet1 = new MyThreadRunnable1();

        Thread gun1 = new Thread(bullet1);

        MyThreadRunnable2 bullet2 = new MyThreadRunnable2();

        Thread gun2 = new Thread(bullet2);

        gun1.start();

        gun2.start();

    }

}

***Life Cycle of a Thread:***

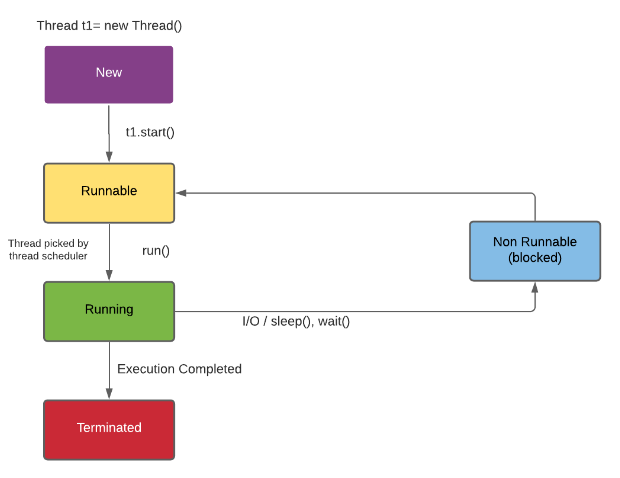
**New->** when instance of a thread is created

**Runnable->** when start() is invoked and before it is selected by the scheduler to be run.

**Running->** after the thread scheduler has selected it.

**Blocked/Waiting->** In this state, the thread is not using CPU time and is waiting for a specific event to occur. It may send a request to a web page for a info which may take some time. By this time other threads will run.

**Termination->** The thread enters the "Terminated" or "Dead" state when it has completed its execution or is explicitly terminated



The ‘super’ keyword refers to superclass (parent) objects. It is used to call superclass methods, and to access the superclass constructor.

MyThread t1 = new MyThread("Harry");//Sets the thread name as Harry.

t1.getName(); t1.getId();

***What is the purpose of the volatile keyword in Java?***

In this example, the flag variable is declared as volatile. This ensures that changes to the flag variable made by one thread are immediately visible to other threads.

public class SharedResource {

    private volatile boolean flag = false;

    public void setFlagTrue() {

        flag = true;

    }

    public boolean isFlag() {

        return flag;

    }

}

***Explain the difference between preemptive scheduling and time-slicing in the context of thread scheduling.***

Preemptive scheduling is when the operating system can forcibly remove a thread from the CPU and give it to another thread. Time-slicing is when each thread is given a certain amount of time to run on the CPU. The main difference is that in preemptive scheduling, the operating system can interrupt a thread at any time, while in time-slicing, the thread is only interrupted when it has used up its allotted time.

***Abstract class:***

Abstract classes provide a way to define a common interface for a group of related classes. any concrete subclass must provide an implementation for those methods if the abstract class. Abstract classes can be used to implement the Template Method Pattern, where the overall structure of an algorithm is defined in the abstract class, and specific steps are left to be implemented by subclasses.

**Abstraction:** Interfaces allow you to define abstract methods without specifying their implementation details. This promotes a high level of abstraction, enabling classes to provide their own implementations while adhering to a common interface.

Subclasses cannot override **static** methods because they are not inherited by the subclasses.

The **static** keyword in Java is like a tag that you put on something (a variable or a method) inside a class. When you mark something as **static**:

1. **For Variables:**
   * Instead of each object (instance) of the class having its own copy of the variable, there is only one copy shared by all instances of the class.
   * You can think of it as a variable that is shared among all objects of the same class.
2. **For Methods:**
   * Instead of being associated with an instance of the class, the method is associated with the class itself.
   * You can call the method using the class name, and you don't need to create an object to use it. Class\_name.fun();
   * It's like a method that belongs to the class and not to any particular object.

the **final** keyword is used to indicate immutability, prevent method overriding, or prevent class inheritance.

***composition*** is a design principle where a class contains an object of another class, rather than inheriting from it. The class that contains the object is referred to as the "composite" or "containing" class. The class whose object is held is referred to as the "component" or "contained" class.

// Component class

class Engine {

    void start() {

        System.out.println("Engine starting...");

    }

}

// Composite class

class Car {

    // Composition: Car has an Engine

    private Engine engine;

    // Constructor

    public Car(Engine engine) {

        this.engine = engine;

    }

    // Method that uses the Engine

    void startCar() {

        System.out.println("Starting the car.");

        engine.start(); // Delegating the start functionality to the Engine object

    }

}

// Main class

public class Main {

    public static void main(String[] args) {

        // Creating an Engine

        Engine myCarEngine = new Engine();

        // Creating a Car with the Engine

        Car myCar = new Car(myCarEngine);

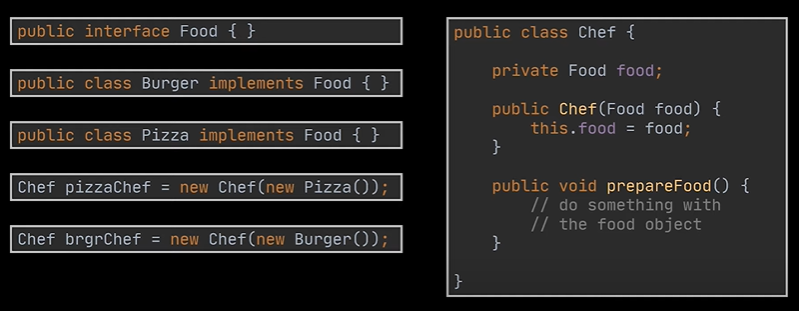
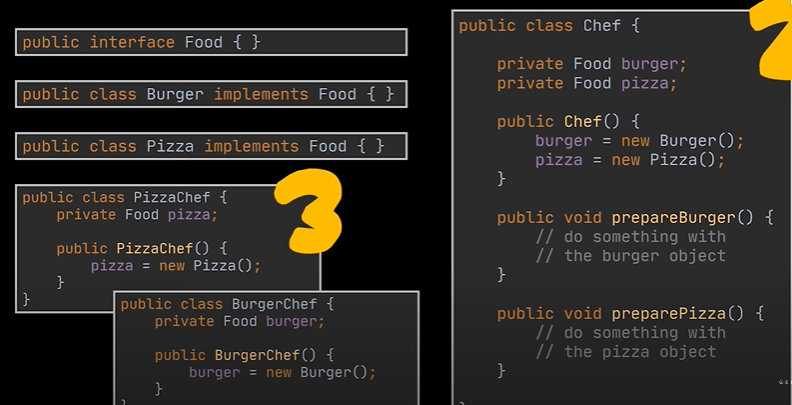
        // Starting the Car, which in turn starts the Engine

        myCar.startCar();

    }

}

***Dependency Injection*** (DI) is a design pattern in Java (and other programming languages) that facilitates loose coupling and enhances the flexibility, maintainability, and testability of software components. Dependency injection involves injecting the dependencies of a class from the outside rather than creating them within the class itself.



1. **Single Responsibility Principle (SRP):**
   * Do one thing and do it well. Each class should have only one job. A class should have only 1 reason to change.
2. **Open/Closed Principle (OCP):**
   * You can add new features to a system without changing the existing code.
3. **Liskov Substitution Principle (LSP):**
   * You should be able to replace a parent class with a child class without messing things up. The methods got from the parent should be meaningful for the child.
4. **Interface Segregation Principle (ISP):**
   * Don't force classes to implement interfaces they don't need. Keep interfaces small and specific. Make atomic.
5. **Dependency Inversion Principle (DIP):**
   * High-level modules (like business logic) shouldn't depend on low-level modules (like data storage). Both should depend on abstractions (like interfaces). Low level modules should have dependencies on higher level module. The implementation should be in lower level module. Class should depend on interface rather than concrete class.

***Git/ GitHub***

Clone a repository to our local repository, make changes and push the updated files

git –version

pwd (Print Working Directory)

cd FileName (change directory)

cd .. (get out of the directory)

ls (list of content of current working directory)

ls –a (list of content of current working directory including hidden files)

the tilde(~) sign signifies that we are in main directory or main folder

git config –global user.name “Imtiaj Aurpon”( global means whatever changes we make it will be by a specific email.)

git config –global user.email “imtiajaurpon@gmail.com”

git config –list (shows the list of what we have configured so far)

git clone <link> (to copy a repository from GitHub from our local repository)

/ (does the auto complete)

git status

Types of files in git

1. untracked files(The files created in the local repository that git doesn’t know).

2. modified (changed but not staged)

3. staged (added new or changed files to the git staging area)

4. unmodified (after commiting a file)

git add <FileName> (adds new or changed files in our directory to the Git staging area)

git add . (adds everything)

git commit –m “message” (record of change)

git push origin main (among the repositories in GitHub, a default repository named origin which has a branch named main. Here the local repository will be uploaded.

Mkdir <Directory\_name> (make a new directory).

git init (create a new git repo)

git remote add origin <-link-> (github repositories are remote repos. We want to make a repository named origin. You are telling Git to associate the remote repository at the provided **<link>** with the name "origin.”

git remote –v (to verify remote)

git branch (to check in which branch we are in. by default master branch)

git branch –M main (rename the branch name to main)

git push –u origin main (it’s a shortcut were push will be done by writing only git push)

git checkout –b new\_branch\_name (create new branch)

git checkout branch\_name (go to another branch named branch\_name)

git branch –d branch\_name (delete the branch named branch\_name)

git diff <-branch\_name-> (shows the difference between the present branch and the branch\_name)

git merge <-branch name-> (merges the two branches)

git pull origin main (used to fetch and download content from a remote repository and immediately update the local repository to match the content in the branch named main here)

git log (to see all the commits)

git reset <-FileName-> (resets the staged changes)

git reset

git reset HEAD~1 (resets for one commit)

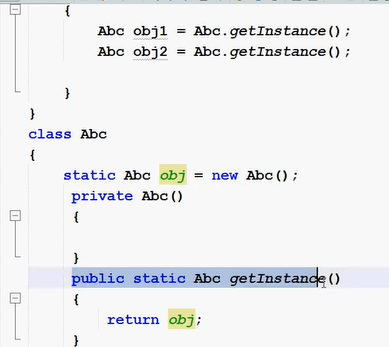
git reset <-commit\_hash-> (resets for many commits in git)

git reset –hard <-commit\_hash-> (resets for many commits in git as well as in VS code)

In GitHub, a "fork" is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. You can make changes to your forked copy, propose those changes to the original repository through pull requests, and contribute to open source projects.

***Singleton design pattern***

A singleton is a design pattern in object-oriented programming (OOP) that restricts the instantiation of a class to a single instance and provides a global point of access to that instance. The purpose of a singleton pattern is to ensure that a class has only one instance and to provide a global point of access to that instance. 1. create an object 2. Make the constructor private 3. Make a function that returns the object.



In Java, the **throws** keyword is used in the method signature to declare that a particular method may throw one or more specified exceptions during its execution. This allows the method to indicate the types of exceptions that callers of the method should be prepared to handle or propagate using try catch block.

public void exampleMethod() throws SomeException, AnotherException {}

In Java, the **throw** keyword is used to explicitly throw an exception.

        try {

            if (someCondition) {

                throw new SomeException("This is a custom error message");

            }

        } catch (SomeException e) {

            System.out.println(e.getMessage());

        }

***Stringbuffer functions***

Append(String str)

Insert(int offset,String str)

Delete(int start,int end)

deleteCharAt(int index)//throws an exception StringIndexOutOfBoundsException

replace(int start,int end,int ,String str) //out of bound will add the string to the end

reverse()

charAt(int index) //throws an exception StringIndexOutOfBoundsException

length()

toString()

***Serialization*** in object-oriented programming (OOP) refers to the process of converting an object's state into a format that can be easily stored or transmitted and later reconstructed. This is particularly useful when you need to persistently store objects, send them over a network, or pass them between different parts of a distributed system.

Kruskal's algorithm is often preferred when the graph is sparse, whereas Prim's algorithm can be more efficient for dense graphs.

1. **Full Binary Tree (Proper Binary Tree):**
   * Every node in the tree has either 0 or 2 children. No node has only one child.
2. **Complete Binary Tree:**
   * A binary tree in which all levels are completely filled except possibly for the last level, which is filled from left to right.
3. **Perfect Binary Tree:**
   * A binary tree in which all levels are completely filled, and all leaf nodes are at the same level.
4. **Balanced Binary Tree:**
   * A binary tree in which the height of the left and right subtrees of any node differs by at most one. This property helps in ensuring that the tree remains relatively balanced, facilitating efficient operations.
5. **Degenerate (or pathological) Tree:**
   * A binary tree where each parent node has only one associated child node. It essentially becomes a linked list.
6. **Binary Search Tree (BST):**

* A binary tree where the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree only nodes with keys greater than the node's key. This structure allows for efficient searching, insertion, and deletion operations.

1. **AVL Tree:**

* A self-balancing binary search tree in which the height of the two child subtrees of any node differs by at most one. Rotations are performed to maintain balance after insertion or deletion.

1. **Red-Black Tree:**

* Another type of self-balancing binary search tree where each node has an extra bit for denoting the color (either red or black). The color is used to ensure that the tree remains balanced during insertion and deletion. **Binary Search Tree Property, The root is black, All leaves (NIL or null nodes) are considered black, Red nodes cannot have red children**

DOM represents the content of XML or HTML document as a tree structure.

