**Core Java Syllabus**

(Core Concepts + GUI + JDBC + New features + Mini Project)

**Core Concepts:**

**Need of Programming:**

* Introduction to Programming
* Types of applications
* Compiler / Interpreter
* History of Java
* JDK versions
* JDK includes
* Certifications
* Platform dependency
* Platform independency

**Introducing Programming:**

* Programming elements
* Introducing variables & methods

**Introducing Object oriented approach:**

* Procedure oriented programming
* Object oriented programming
* Object and Class
* Structure of Java application
* Contexts of Java application(Static and Non static)

**First Java application:**

* Writing First application(using notepad)
* Installing JDK
* Path settings(Environment variables)
* Path and Classpath
* Compile and Run using Command Prompt

**IDEs:**

* What is IDE
* Basic IDES of Java
* EDIT PLUS configurations
* High End IDES
* Compile and Run Java application using IDES

**Static members**

* Static main() method
* Static block
* Static variable
* Static user method

**Non static members**

* Non static block
* Constructor
* Non static variable
* Non static method

**Data types:**

* Introduction
* Limits
* ASCII and UNICODE
* Type casting

**Operators:**

* Arithmetic operators
* Relational operators
* Unary operators
* Assignment operators.
* Logical operators
* Bitwise operators
* Shift operators
* Ternary operator

**Control Statements:**

* Conditional control statements
  + If statement
  + If else
  + If else if
  + Nested if
  + Switch
* Loop control statements
  + While loop
  + For loop
  + Do-while loop
  + For each loop
  + Nested loops
* Branching statements
  + Break
  + Continue
  + Return

**Access Modifiers**

* Introduction
* Accessing private members
* <package> modifier
* Protected members
* Public members

**Packages**

* Introduction
* Import statement
* Java.util.Random class
* Java.util.Scanner class
* Create user package
* Connecting classes of same package
* Connecting classes of different packages
* User defined System class
* Sub packages creation

**Wrapper classes**

* Primitive types v/s Wrapper classes
* Primitive to Object (Boxing)
* Object to Primitive (Unboxing)
* Primitive to String
* String to Primitive
* Object to String
* String to Object
* Fields of Wrapper classes

**Command Line Arguments**

**Object Oriented Programming:**

* Introduction
* Object & Class
* Accessing Static and Non static members
* Get() and Set() methods
* This keyword
* This() method
* **Encapsulation**
* **Inheritance**
  + Introduction
  + Types of Inheritance
  + Single inheritance
  + Object creation process
  + Method overriding
  + Multi level inheritance
  + Hierarchical inheritance
  + Super keyword
  + Super() method – Object initialization
* Final keyword
* **Abstraction**
* Interfaces
* **Polymorphism**
  + Compile time polymorphism
  + Runtime polymorphism
* Composition
* Aggregation

**Exception Handling**

* Introduction
* Exception codes
* Hierarchy of Exception classes
* Handling Exception
* Try with multiple catch blocks
* Finally block
* Throw keyword
* Custom exceptions
* Throws keyword

**Multi threading**

* Introduction
* Thread life cycle
* Single threaded application
* Multi threaded application
* Sleep() method
* Join() method
* Thread identities
* Execution time of threads
* Thread synchronization
* Thread pool
* Deadlocks
* Daemon threads
* Runnable interface

**Garbage Collection**

* Introduction
* System.gc()
* Runtime class
* Runtime.gc()
* JVM memory

**Inner classes**

* Introduction
* Static inner classes
* Non static inner classes
* Local inner classes
* Anonymous inner classes
* Create thread using Runnable

**String Handling**

* Introduction
* String Class methods
* Immutability of Objects
* String Buffer
* String Builder class
* String Tokenizer class
* String programs

**Reflection API**

* Introduction
* Finding Object information
* Class.forName()
* Instantiation of class Dynamically

**Java IO**

* Introduction to IO
* Byte streams
* Character streams
* Buffered streams
* Object streams
  + Serialization
  + De-Serialization
* Array streams
* Data streams

**File Handling**

* Introduction
* Create file
* Create directories
* File operations
* Directory operations

**Regular Expressions**

**Collections**

* Introduction
* Arrays
* Arrays v/s Collections
* Generics
* Annotations
* Java Enum
* List interface
  + ArrayList
  + Vector
  + LinkedList
* Set interface
  + HashSet
  + LinkedHashSet
  + TreeSet
* Map interface
  + HashMap
  + LinkedHashMap
  + TreeMap
* Queue interface
  + LinkedList
  + PriorityQueue

**AWT and SWINGS**

* Introduction
* Applets v/s Swings
* Creating Custom JFrames
* Creating Components
* Event Listeners
* Anonymous listener implementation
* Calculator program

**JDBC**

* Introduction
* Drivers
* Java – Database connection
* ORACLE – statements
* Executing DDL Queries from Java program
* DML commands execution
* Transactions program

**Internationalization**

**Java 8 features**

* forEach() method in Iterator interface
* Default and Static methods
* Functional interfaces and Lambda expressions
* Stream API
* Time API

**Java 9 features**

* JShell
* Private methods in interfaces
* Try with resources
* Diamond operator for anonymous inner class

**Java 11 features**

* Downloading JDK11
* Running Java File with single command
* Local variables syntax for lambda expressions

**Mini Project using SWINGS and JDBC**

* Reading information from Database table
* Storing the data temporarily into Map object
* Display the information using GUI

**Core Java**

* Java is a programming language.
* Core java covers only core concepts of Java
* It is called J2SE(Java Software Edition)
* Using Core concepts, we can develop only system based applications.

**What is Programming language?**

* Communication is the concept of sharing information.
* We use a language to share the information.
* To communicate with a machine, programming language is required.
* **Programming language is a pre-defined “software”**

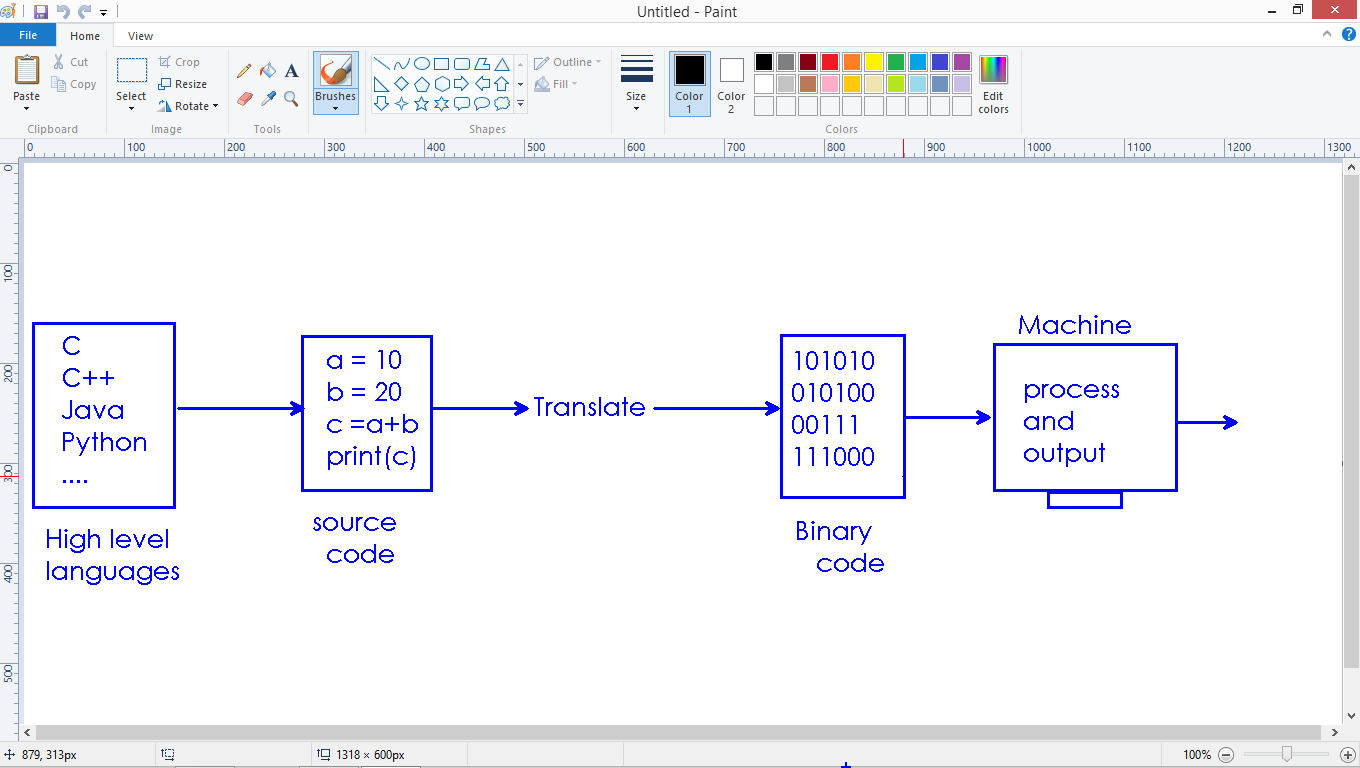
**Need of programming language?**

Trainer ----------🡪 zoom.us ------🡪 Students

* To communicate we use applications (whatsapp, facebook, zoom.us, banking…)
* We can develop application using “programming languages”

**Communication with machine:**

* Using programming languages, we can define source code easily.
* We convert the source code into machine code using translators(compiler and interpreter)
* We can input only binary code to machine process.





**Applications:**

* We use programming languages to develop applications.
* Applications mainly classified into 2 types
  + System level applications – Standalone
  + Network level applications – web apps

**Standalone applications:**

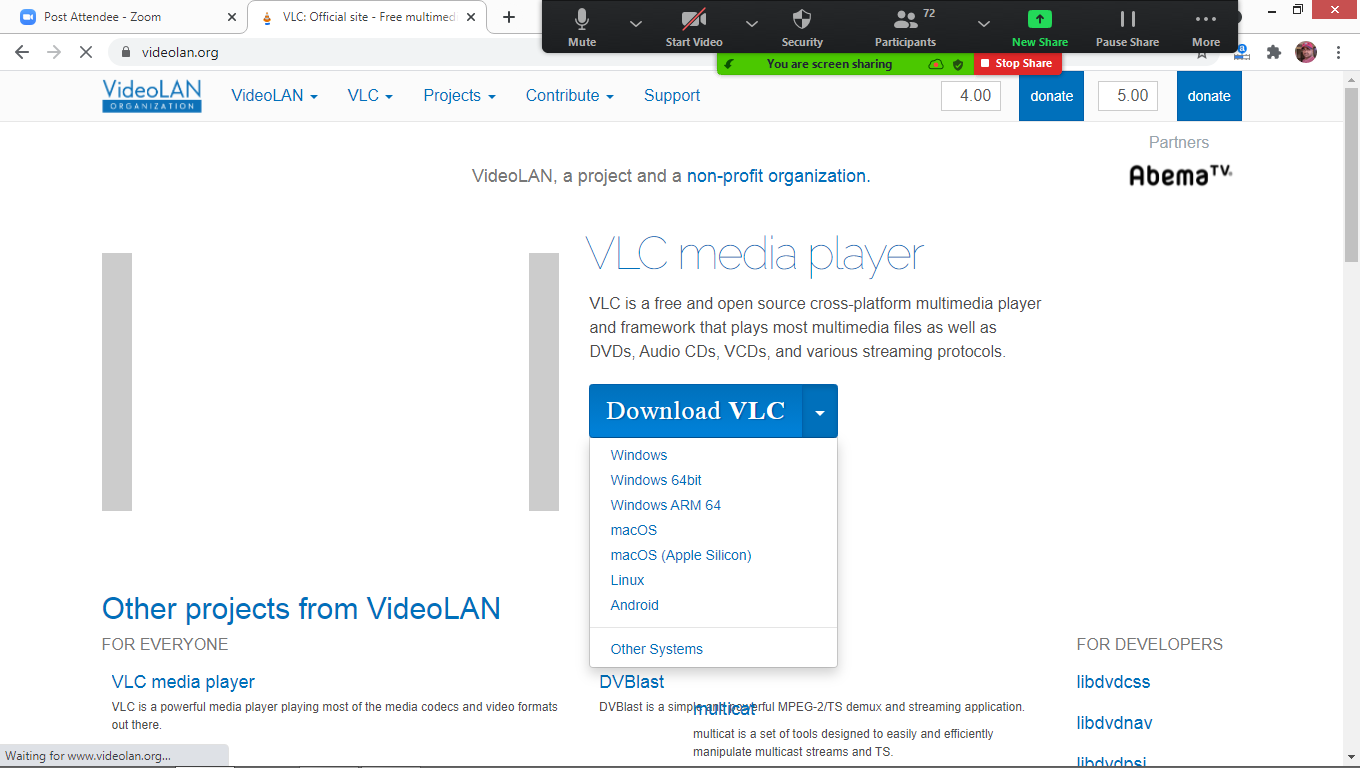
* The application run from single machine
* We must install into system
* Application dependent to Operating system.
* Examples : VLC, Ms-Office, Browser, C, C++, Java, Python….

Windows - .exe files

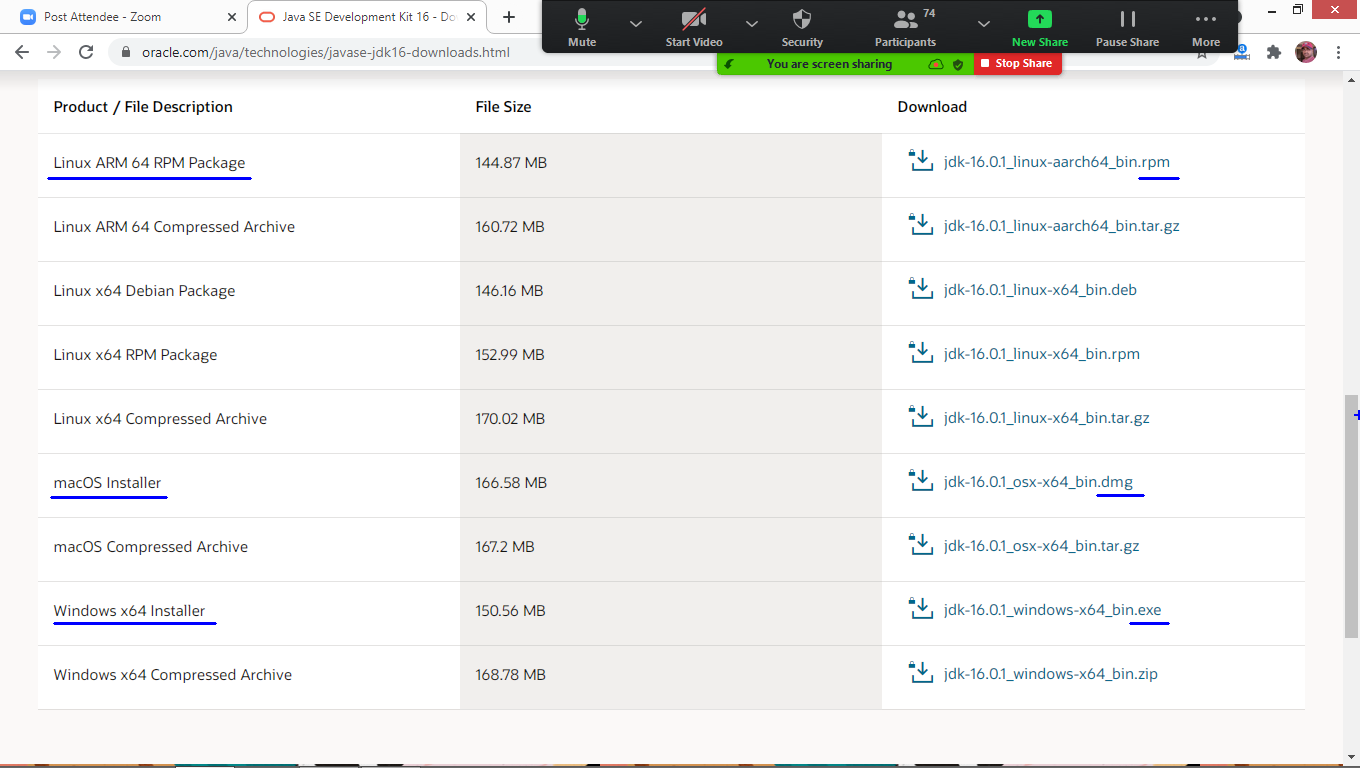
Linux - .rpm files

Mac - .dmg files

Downloading VLC: check the OS compatability while downloading

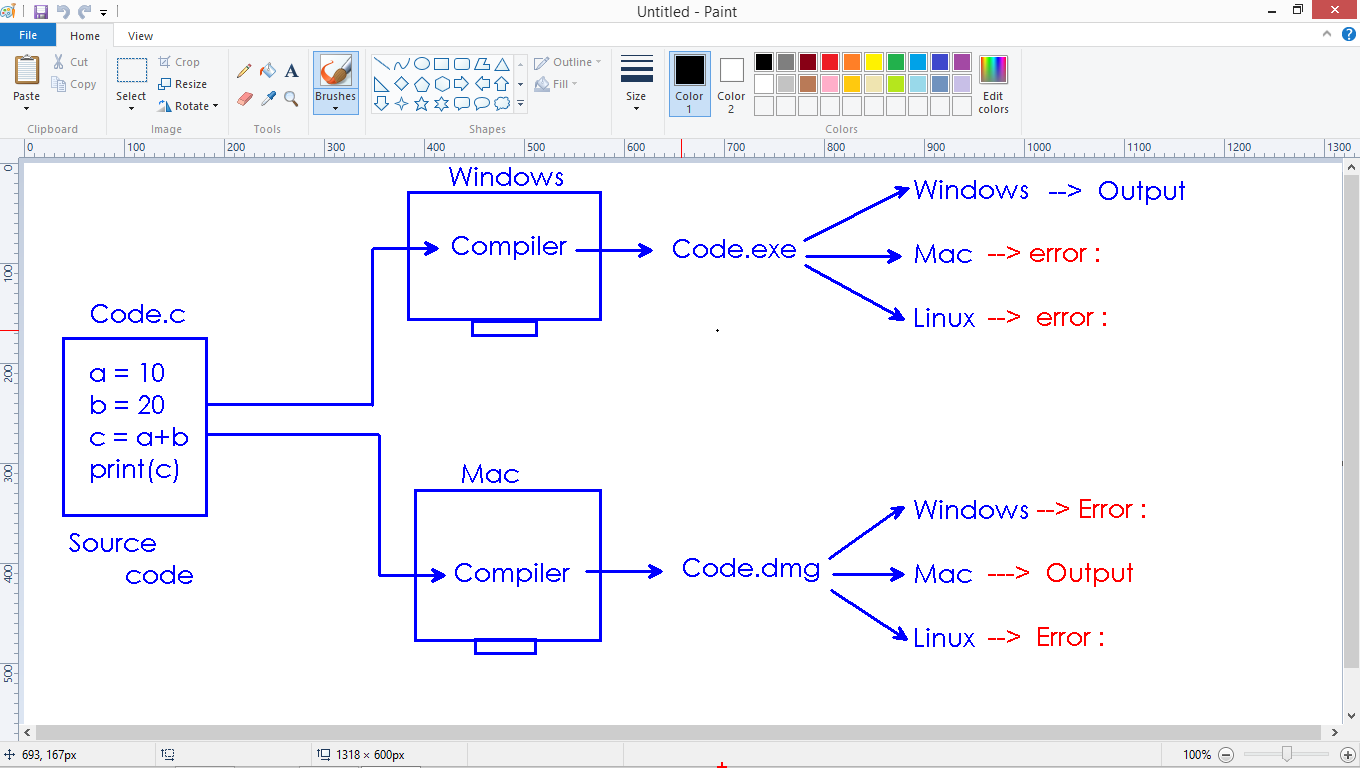


Downloading Java: check the OS compatability while downloading



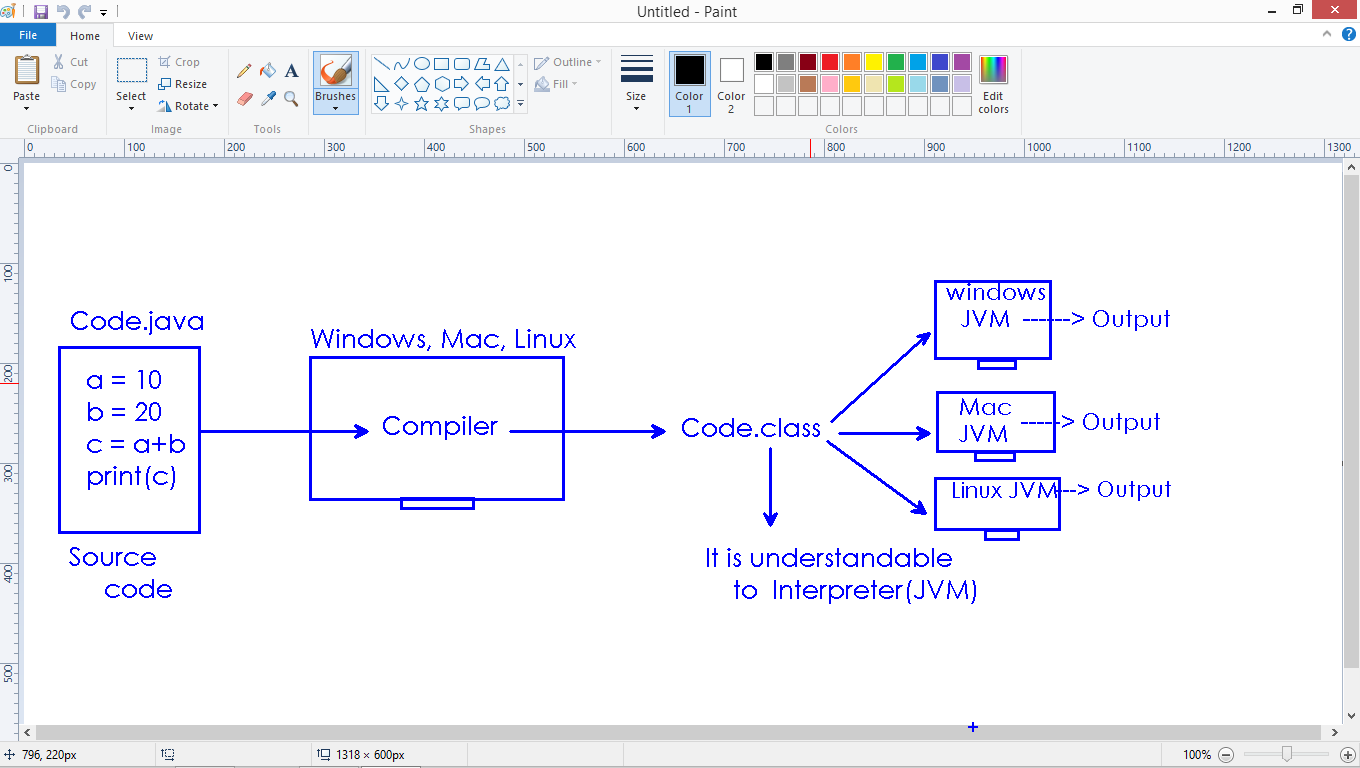
**Platform dependency:**

* C, C++ applications are platform dependent.
* Using these languages, we can develop only Standalone applications.
* C compiler converts the source code into specific OS understandable instructions.
* Compiled code is compatible to same OS only.

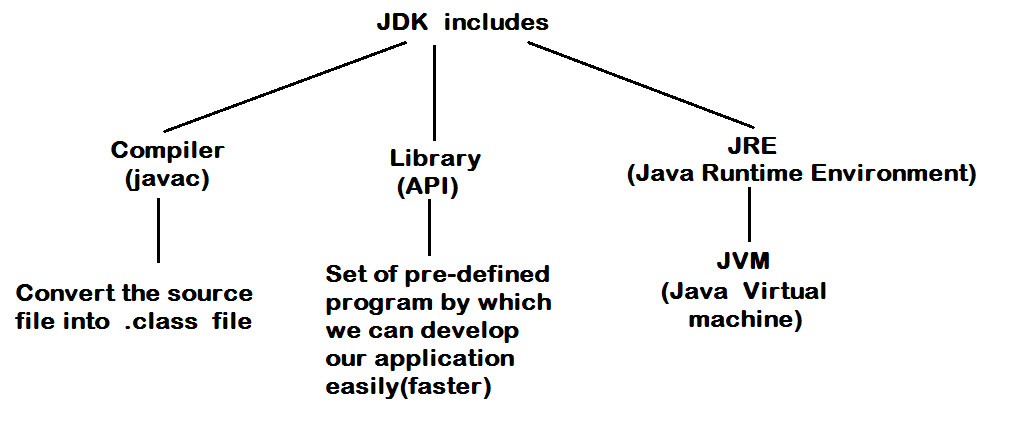


**Platform independency:**

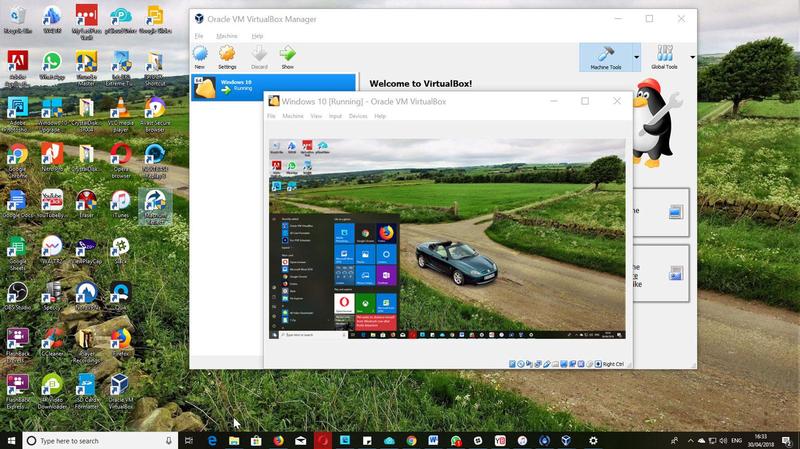
* Java, .Net and Python are platform independent programming languages
* Using these languages we can develop both standalone and web applications.
* Java Compiler converts the source code into .class file
* .class file is understandable to Java interpreter only.
* Java interpreter run the .class file and gives the output on any OS.

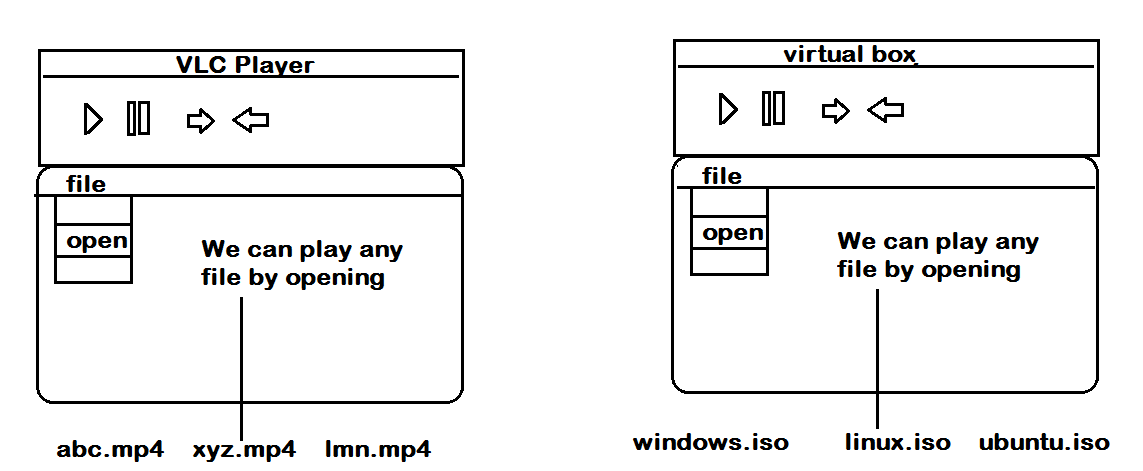


**Java Includes:**



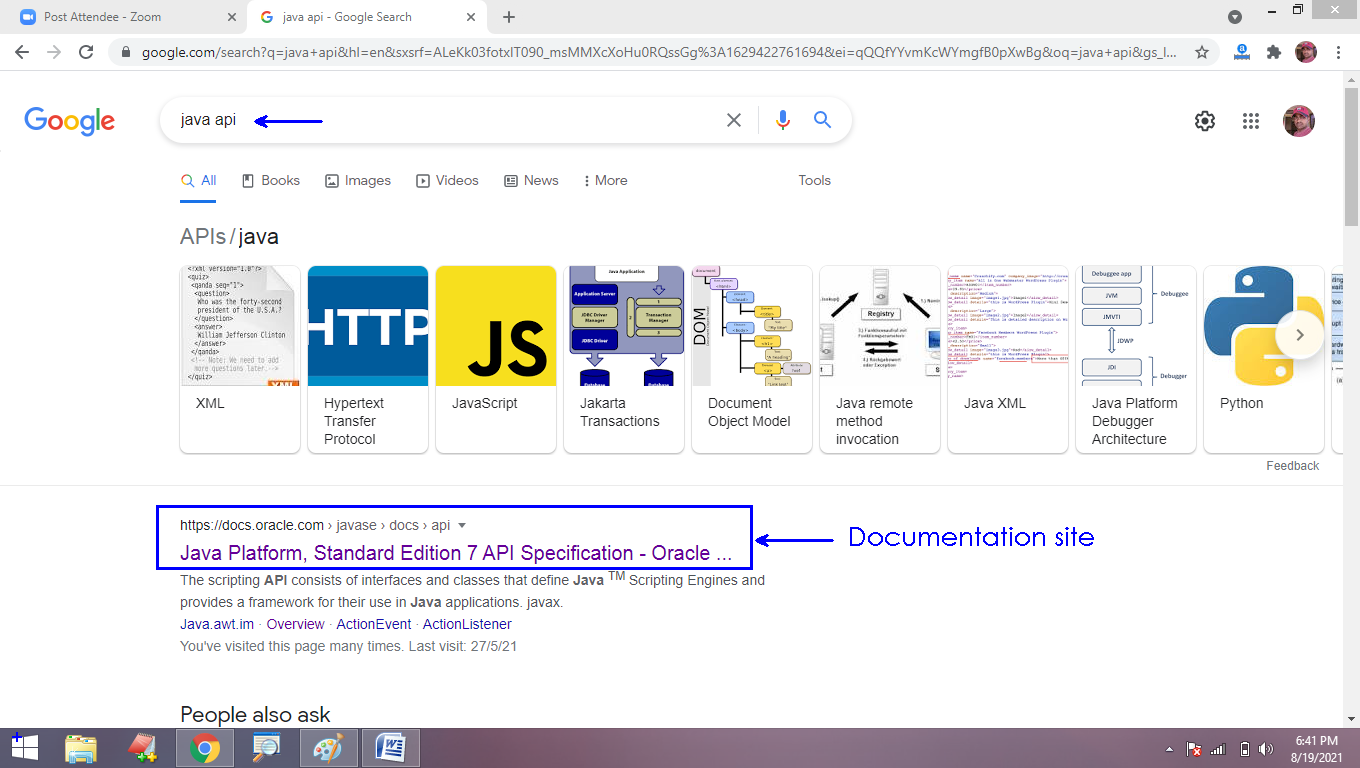
**Virtual Environment:**

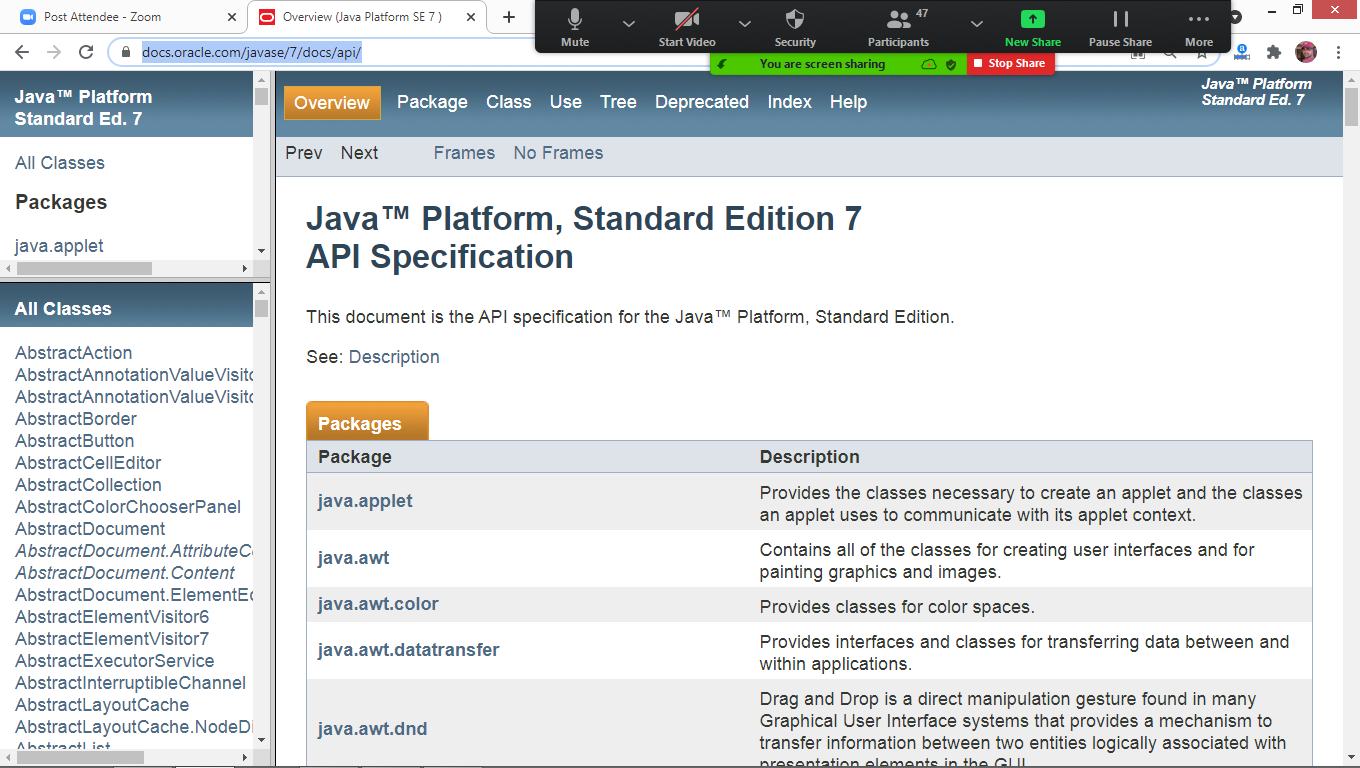


****

**Java API Documentation:**

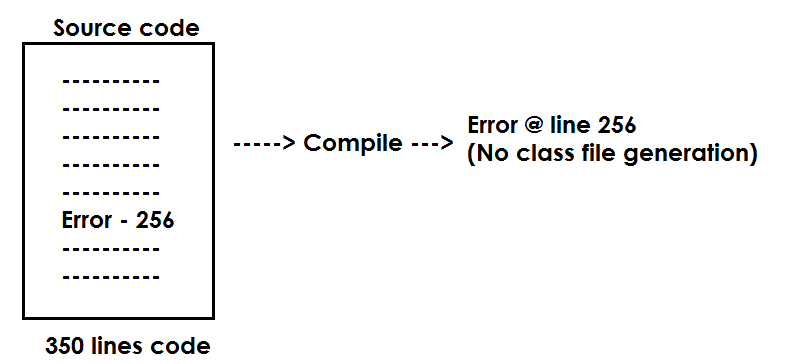
* Java developers written many programs and provided as Library
* Library is technically called “Java API”
* While using any program in our application, we have to study about that program.
* The documentation(theoretical explanation) of program available as Java API documentation.
* It is an official website.
* <https://docs.oracle.com/javase/7/docs/api/> is giving information about library.

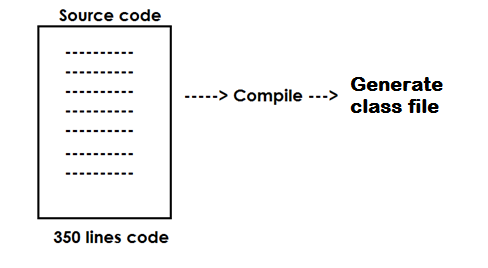




**Compiler:**

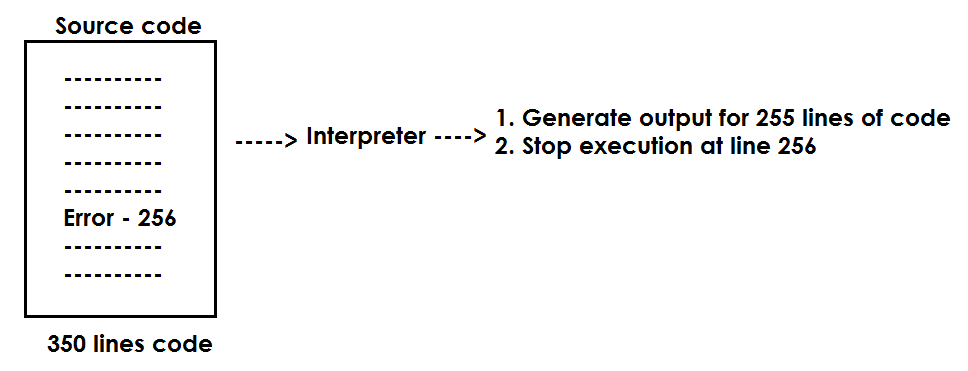
* It is a program.
* It checks the program is syntactically correct or not (following programming language rules or not).
* Compiler converts all instructions are correct, then it converts the source file into .class file
* Compiler is called(All at once translator).





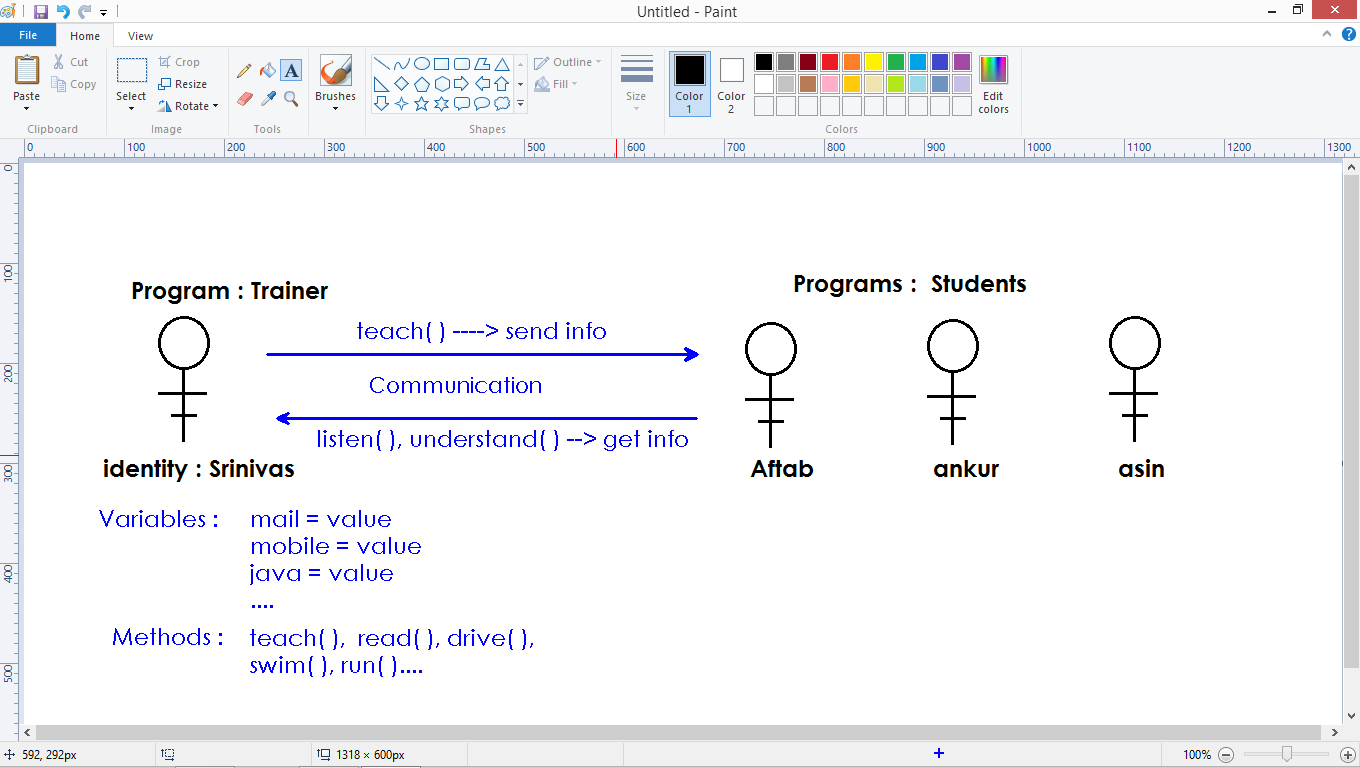
**Interpreter:**

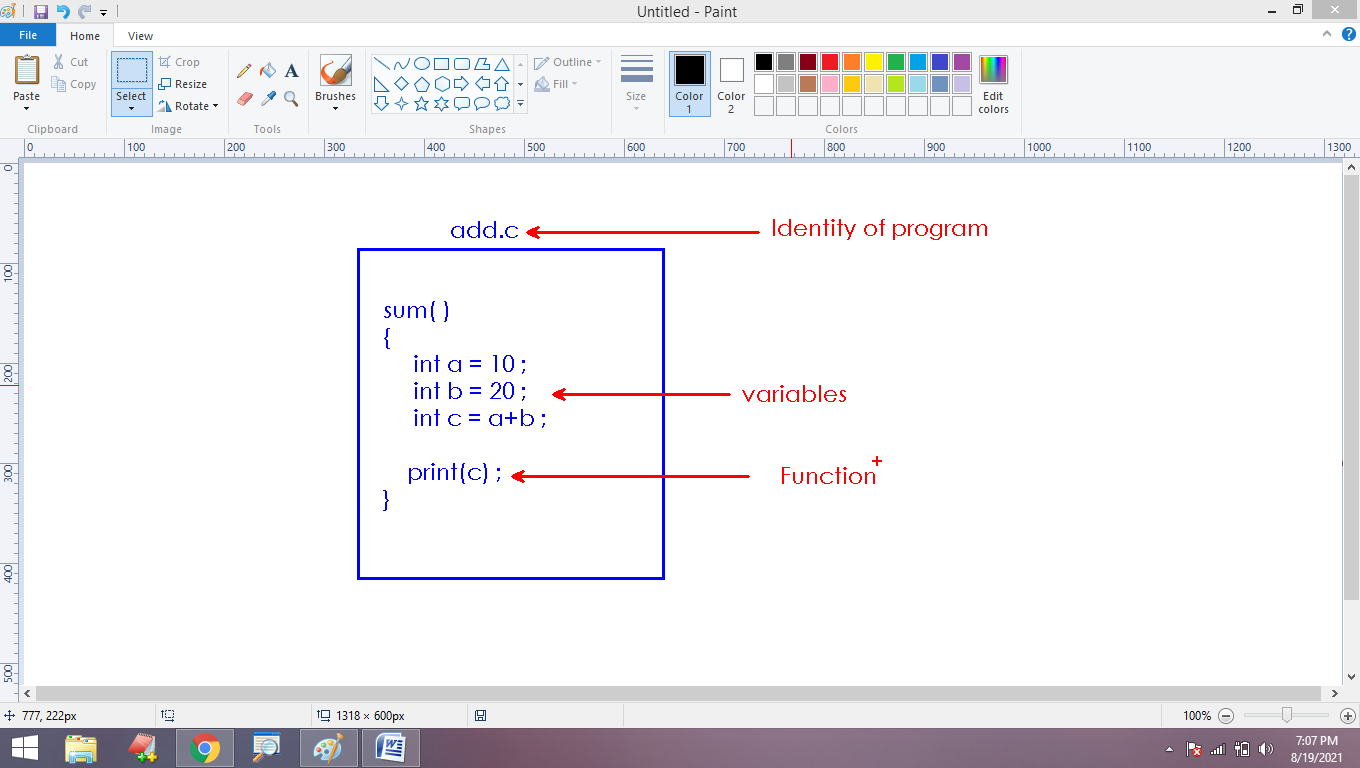
* It is a program
* It translates the code line by line

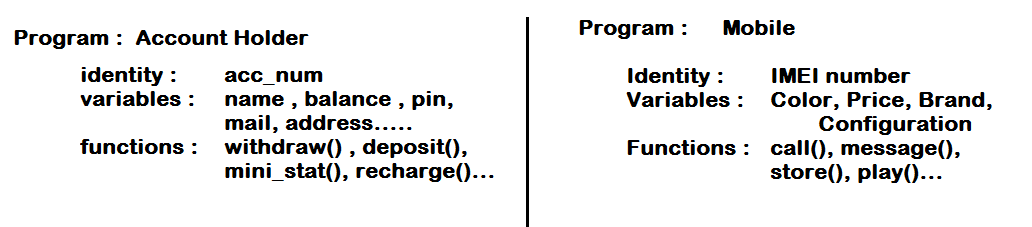


**Programming elements:**

* Programming languages are used to develop application.
* Application(software) is a set of programs
* Every program consists
  + Identity : Every program has unique identity by which other program will call for communication.
  + Variables : are used to store the information.
  + Methods : are used to communication. We share the information in communication.

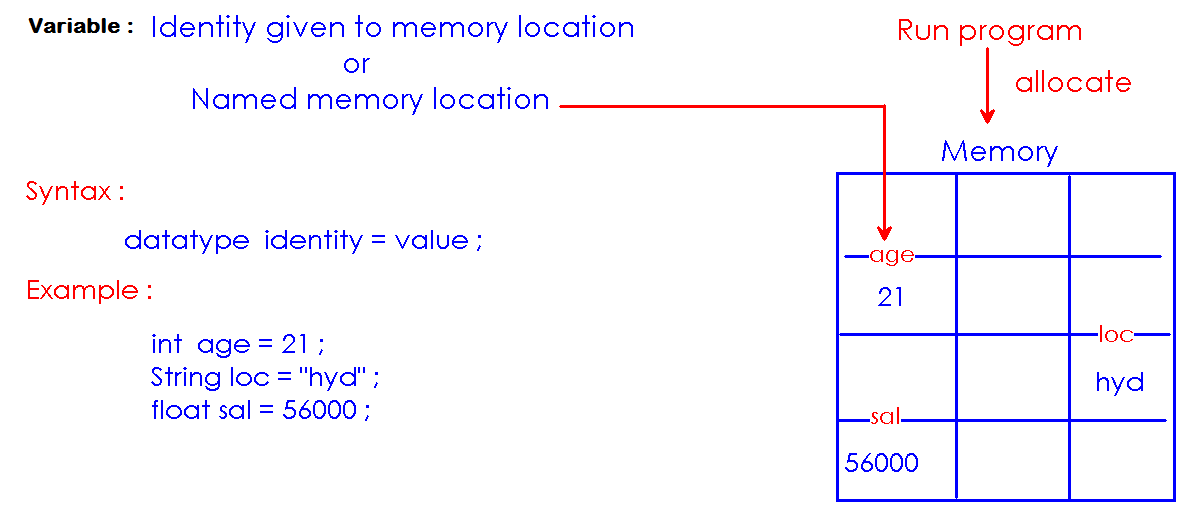






**Variable:**

* Identity of memory location
* It is used to process the information of memory location.

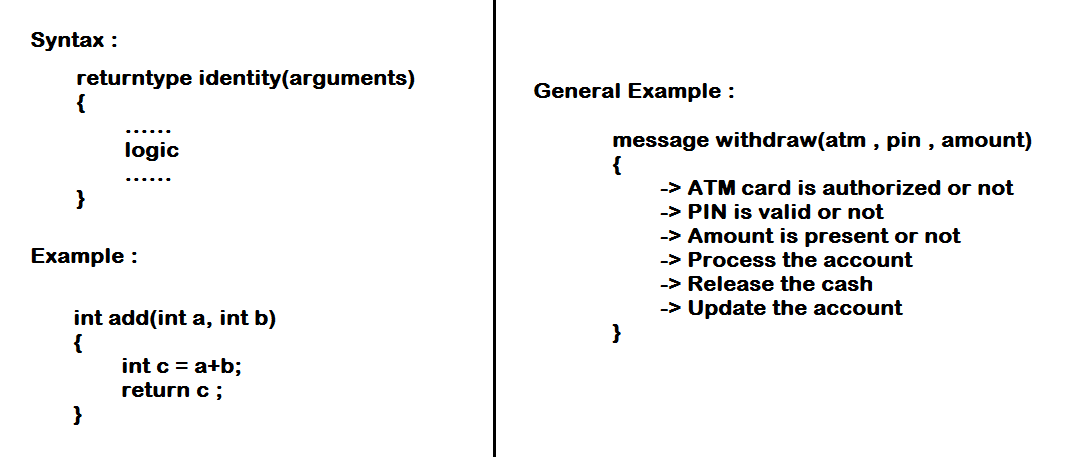


**In Java, variables are classified into:**

1. Local variables
2. Static variables – Class level
3. Instance variables – Object level

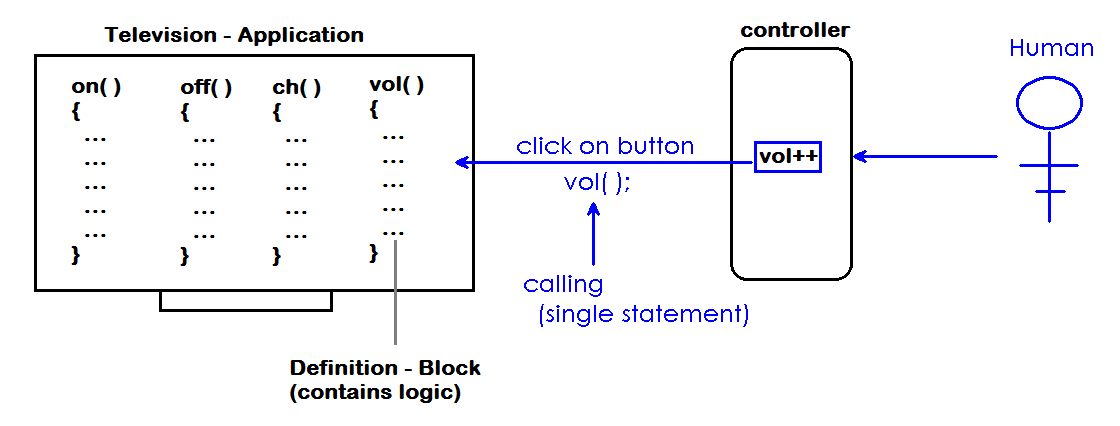
**Function or Method:**

* A block of instructions with an identity.
* Function takes input(arguments), process the input and generate the output.



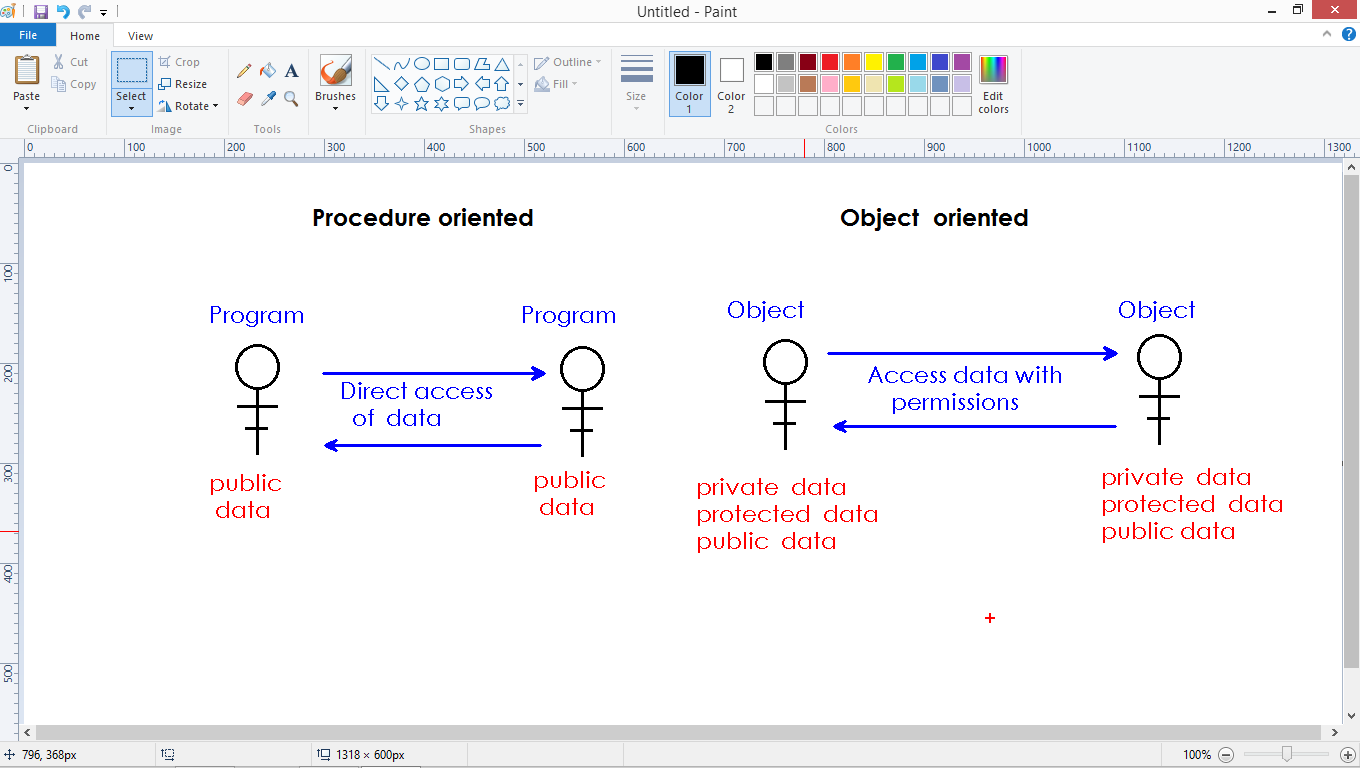
**Every method consists:**

1. Definition: It is a block of instructions contains the logic to perform the task.
2. Callng : It is a single statement and is used to access that logic



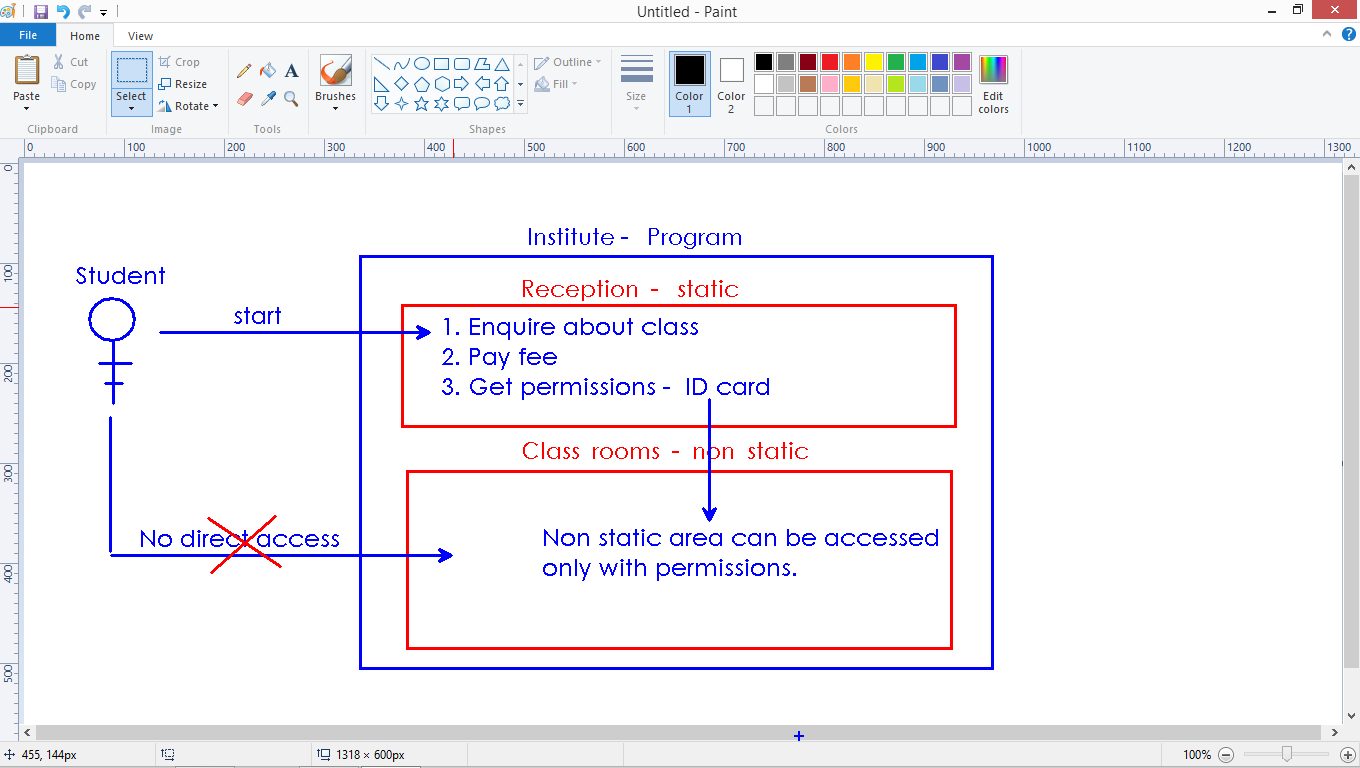
**Procedure Oriented v/s Object oriented programming:**

* C language is procedure oriented
* Java is Object oriented programming language.
* In procedure oriented app, one program can access the information of another program directly. Hence less security
* In Object oriented app, one program cannot access the information directly.
* In Object oriented programming, data is protected.
* We protect the data using “access modifiers”



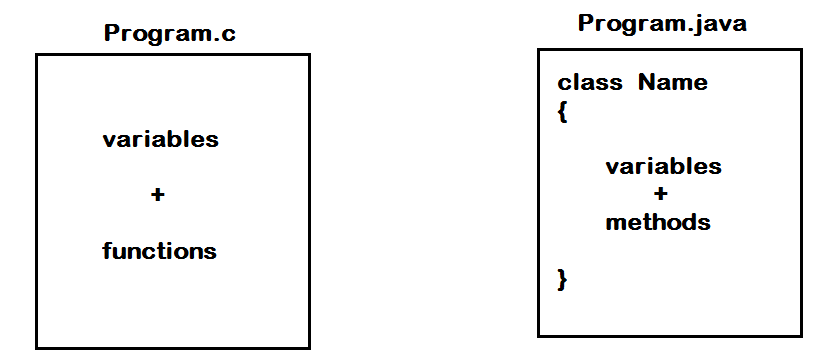
**Application contexts:**

* Every java application has 2 contexts
  + Static – Free access
  + Non static – Restricted access



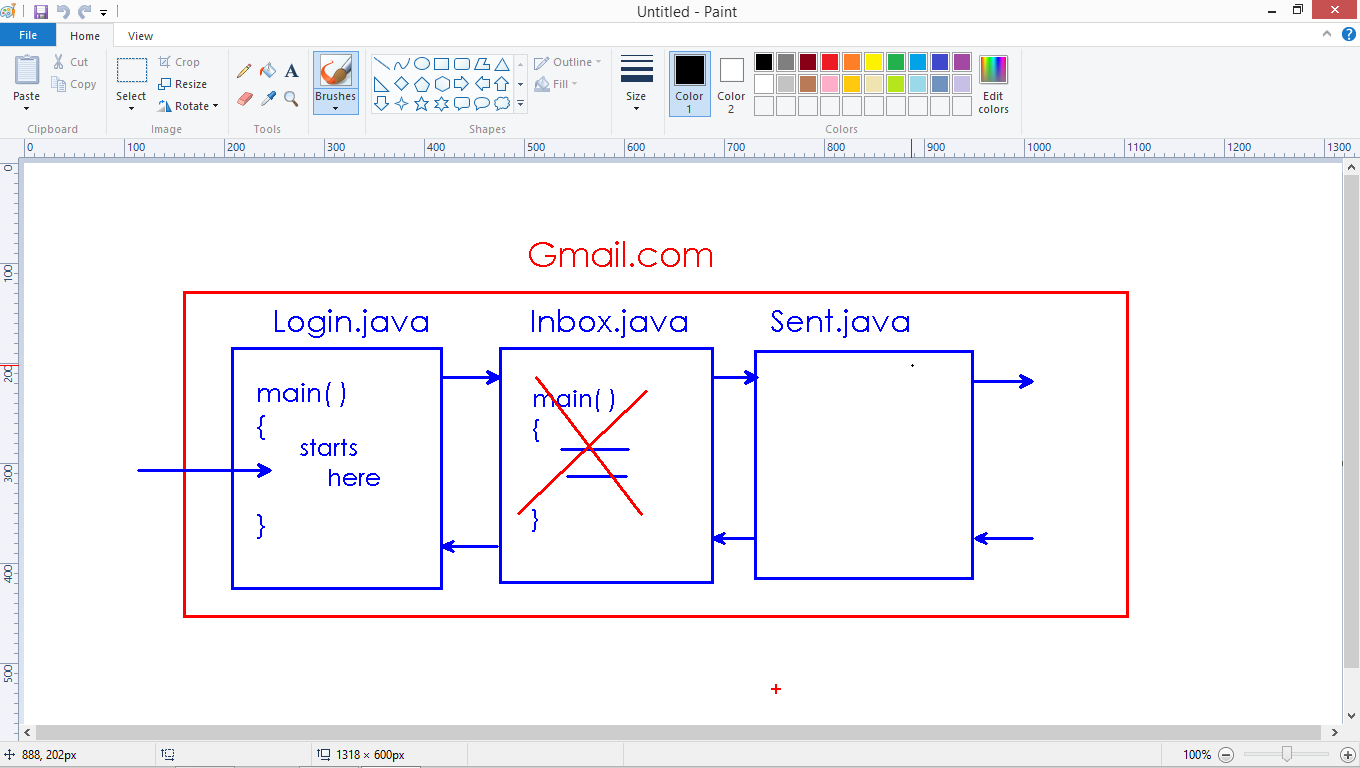
**The way of writing program in procedure oriented app and object oriented app:**

* In Object oriented application, every instruction must be placed inside the class.
* Class represents Object.

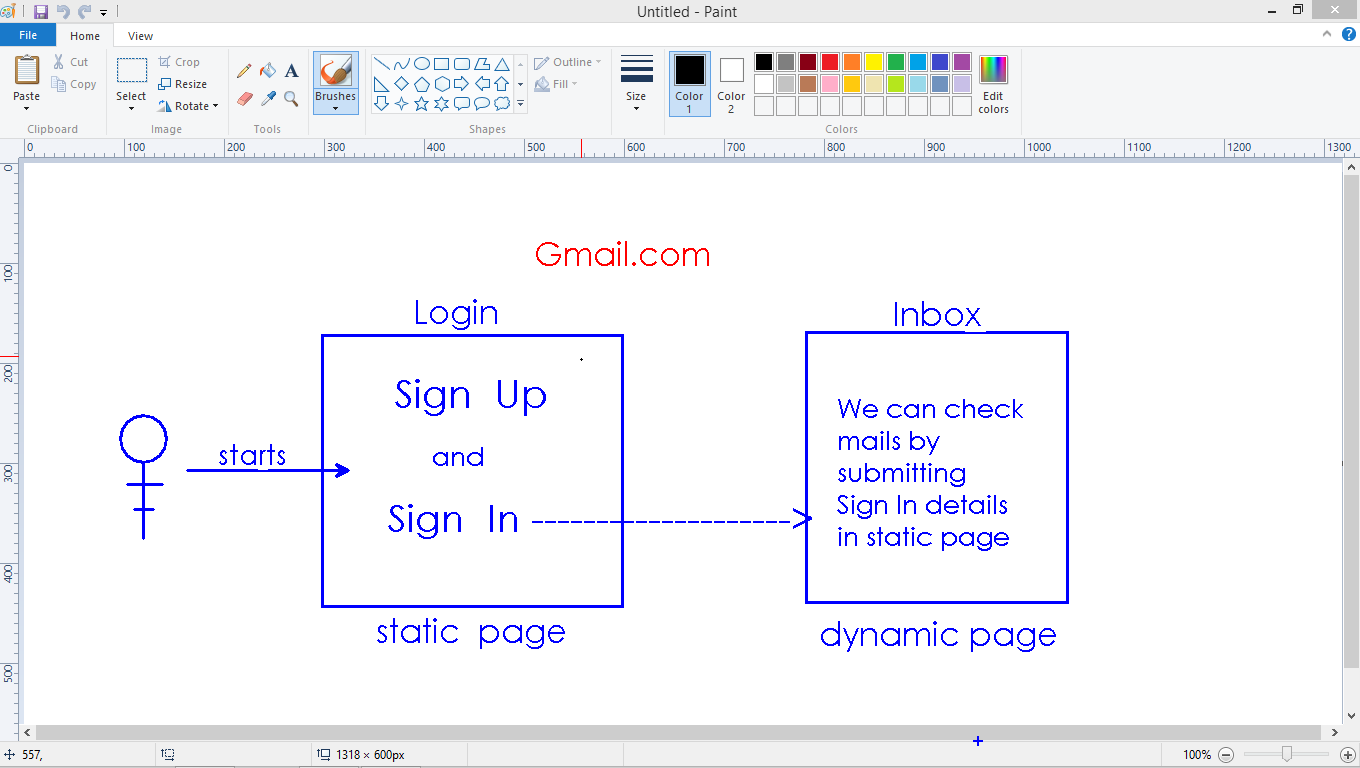


**Structure of Java application:**

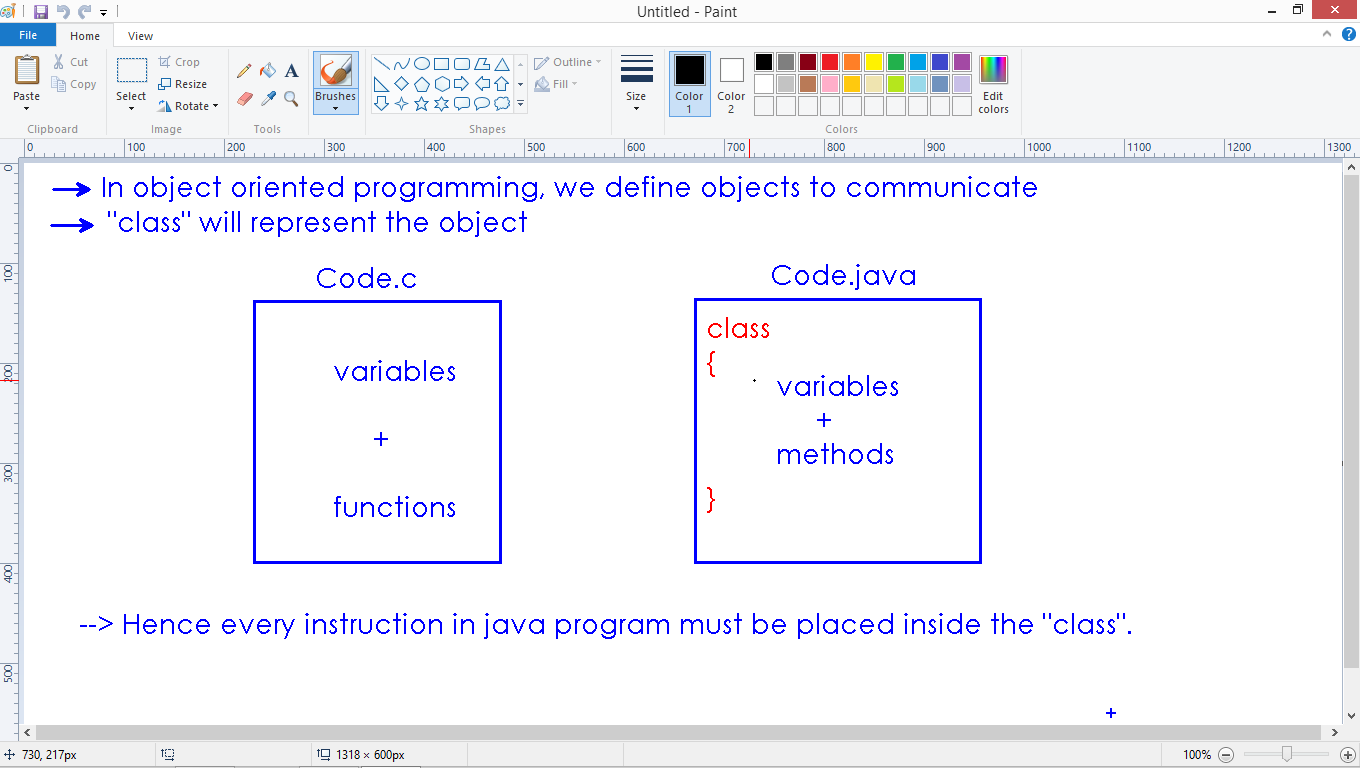
* Application is a collection of programs
* Java application is a collection of source files(.java files / .class files)
* Every application has a single starting point – main() method.
* Main() method must be placed in a single file by which application execution starts.



* We define static and non static programs in application
* Program execution always starts with “static context”
* Hence main() method is static – it is starting point
* To access non static members – permissions required.



Writing programs in procedure oriented and object oriented programming:



**Naming conventions:**

* While writing class names, variable names, method names and other members in java application, we must follow some naming rules.
* Based on naming rules, the java code become more readable and understandable.

**class :** Camel Case and No space

String

PrintStream

NullPointerException

FileNotFoundException

ArrayIndexOutOfBoundsException

**method :** Mixed case and no space

main( )

getName( )

getAccountHolderNumber( )

**variable :** Mixed case and no space

int sum = 0 ;

float shoeSize = 8.9 ;

**constant variable :** All caps seperated with underscore( \_ )

MIN\_VALUE = 0 ;

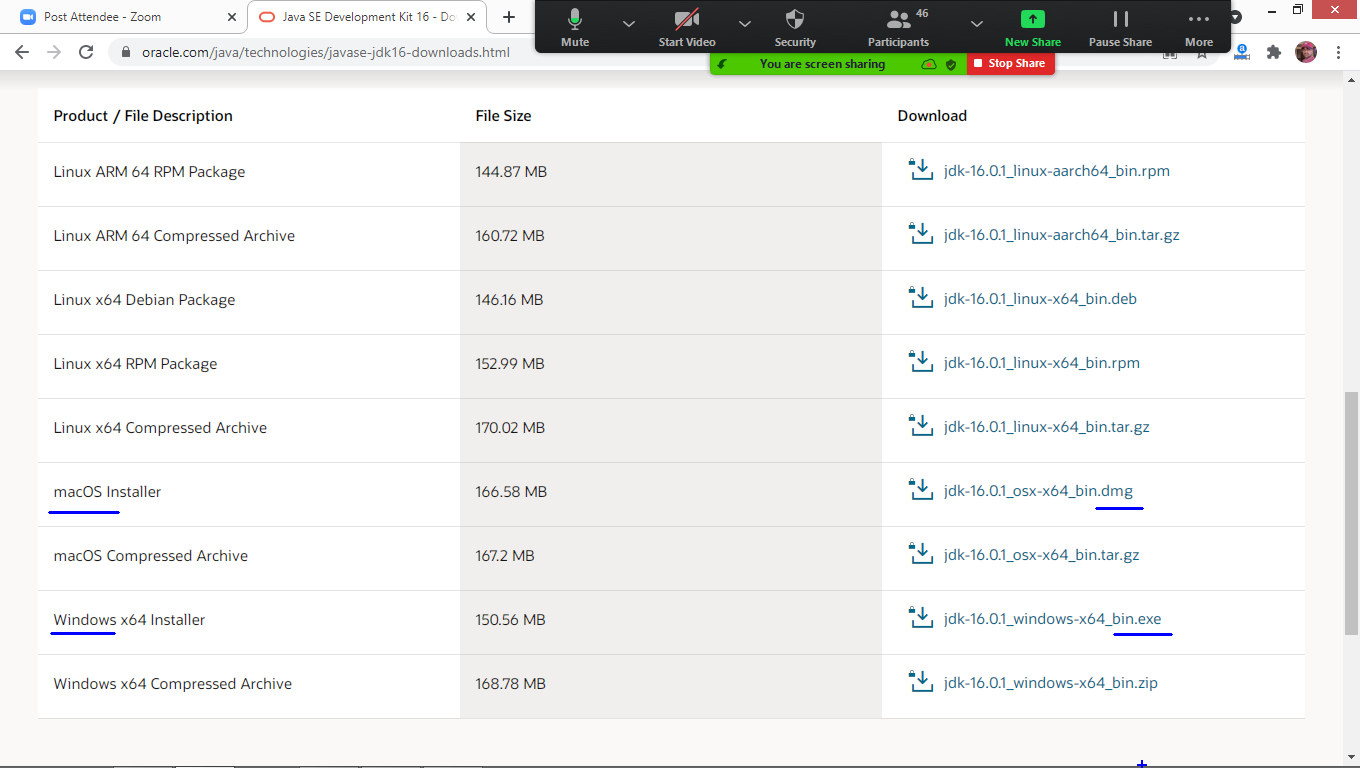
MAX\_VALUE = 10 ;

**Installing JDK:**

* We can download the latest version from official website :

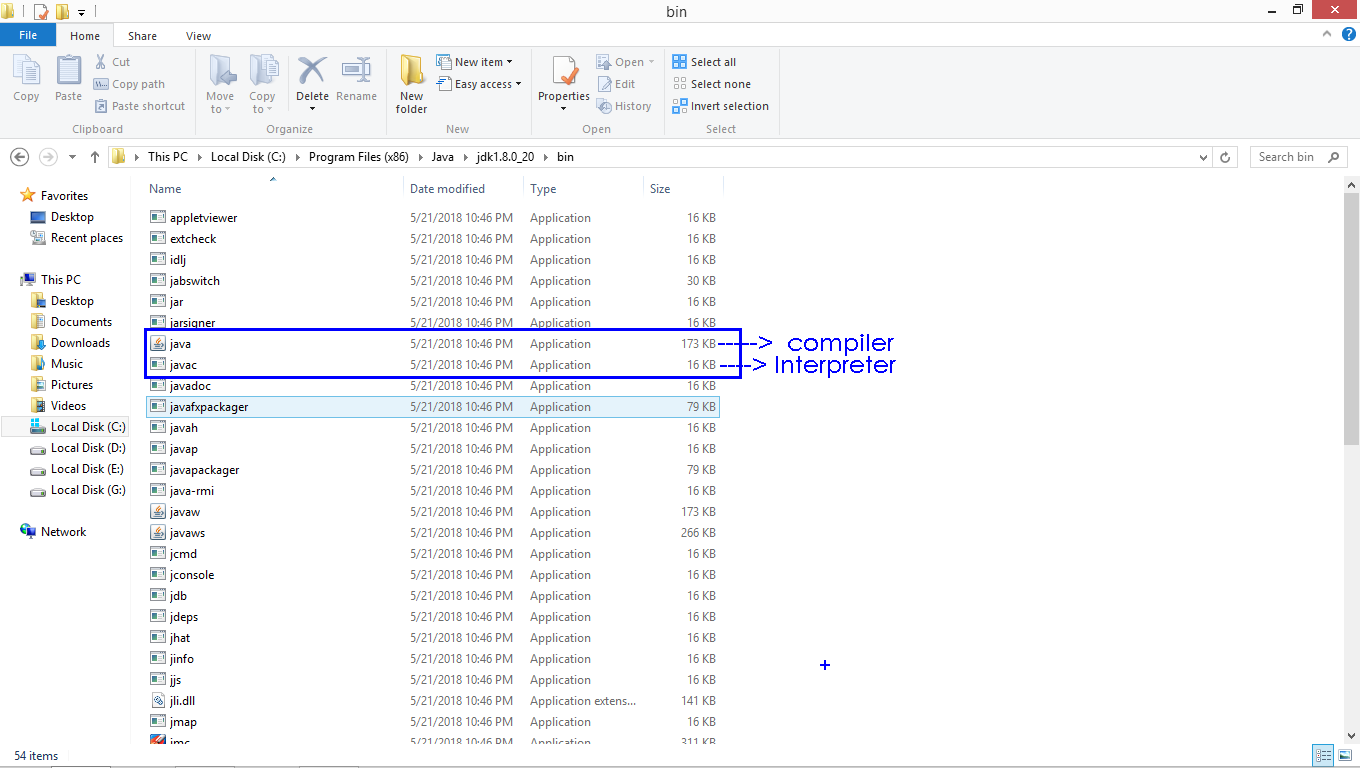
<https://www.oracle.com/java/technologies/javase-jdk16-downloads.html>

* Download the compatible version from the specified list



Run the file(double click) after download

* After installation of JDK, we use 2 commands(programs) in the java installed path.
  + javac( java compiler )
  + java ( java interpreter )
* Path is : C:\Program Files (x86)\Java\jdk1.8.0\_20\bin



**Static Members in Java**

* Every java application has 2 contexts
  + Static context
  + Non static context
* Static context – is free accessible area in java application
* Non static context – is restricted – we can access with permissions
* Program execution(communication with application) is always starts with static area.
* Static Members are :
  + Static main() method
  + Static block
  + Static variable
  + Static method

main():

* Every application execution starts with main() function.
* Java interpreter execute java program
* Java interpreter implicitly invokes main() method.
* Main() is static – free access – starting point of application.

class Code

{

public static void main(String[] args)

{

System.out.println("Hello World!");

}

}

class Program

{

public static void main(String args[])

{

int a=10, b=20;

System.out.println("a value is : " + a);

System.out.println("b value is : " + b);

}

}

class Program

{

public static void main(String args[])

{

int a=10, b=20;

System.out.println("a : " + a + " , b : " + b);

}

}

class Program

{

public static void main(String args[])

{

int a=15, b=25;

int c=a+b;

System.out.println("Sum of " + a + " and " + b + " is " + c);

}

}

class Program

{

public static void main(String args[])

{

int a=15, b=25;

int c=a+b;

System.out.println(a + " + " + b + " = " + c);

}

}

class Program

{

public static void main(String args[])

{

String name="Komal", loc="Maharashtra";

int age=22;

System.out.println(name+ " age is "+age+" and come from the location "+loc);

}

}

class Program

{

public static void main(String args[])

{

String name="Komal", loc="Maharashtra";

int age=22;

System.out.println(" ' " + name + " ' age is ' " + age + " ' and come from the location ' "+ loc + " ' ");

}

}

**Static block:**

* A block of instructions specified with “static” keyword.
* No identity for static block
* We cannot access blocks explicitly – execute automatically.
* **Note:** Static block executes before main() method.

class Code

{

public static void main(String[] args)

{

System.out.println("Main method...");

}

static

{

System.out.println("Static block");

}

}

* We can define more than one static block in a class.
* All these static blocks execute in the defined order.
* Main() executes after all static blocks execution.

class Code

{

static

{

System.out.println("Static block - 1");

}

public static void main(String[] args)

{

System.out.println("Main method...");

}

static

{

System.out.println("Static block - 2");

}

}

**Local variable:**

* Defining a variable inside the block or method.
* We can access local variable within that block only.
* **Note:** Local variable must be assigned with value before use.

class Code

{

public static void main(String[] args)

{

int a ;

System.out.println("a value : " + a); **// Error : 'a' is not initialized**

}

}

class Code

{

public static void main(String[] args)

{

int a = 10;

System.out.println("a value : " + a);

}

}

**Local variable can be accessed within that block only:**

class Code

{

static

{

int a=10;

System.out.println("Block : " + a);

}

public static void main(String[] args)

{

// int a=20;

System.out.println("Main : " + a);

}

}

**Static variable:**

* Defining a variable inside the class and outside to methods and blocks.
* We specify variable with “static” keyword.
* Static variables also called “Class level variables”
* We access static variables using class name.

class Code

{

static int a=10;

static

{

System.out.println("In Block : " + Code.a);

}

public static void main(String[] args)

{

System.out.println("In Main : " + Code.a);

}

}

* Static variables and local variables can have the same.
* We access static variables – using ‘class name’
* We access local variables – without ‘class name’

class Code

{

static int a=10;

static

{

int a=20;

System.out.println("In Block : ");

System.out.println("Local a : " + a);

System.out.println("Static a : " + Code.a);

}

public static void main(String[] args)

{

int a=30;

System.out.println("In main : ");

System.out.println("Local a : " + a);

System.out.println("Static a : " + Code.a);

}

}

When we access a variable directly with in block, first it is looking for local variable.

If the local variable is not present, it access static variable.

class Test

{

static int a = 10;

public static void main(String[] args)

{

System.out.println("a value : " + Test.a);

System.out.println("a value : " + a);

}

}

class Test

{

static int a = 10;

public static void main(String[] args)

{

int a = 20 ;

System.out.println("static a value : " + Test.a);

System.out.println("local a value : " + a);

}

}

**Default values:**

* Static variables automatically initializes with default values
* int – 0
* float – 0.0
* char – blank
* boolean – False
* pointer(class) – null

class Test

{

static int a ;

static float b ;

static char c ;

static boolean d ;

static String e ; // String = char\*

public static void main(String[] args)

{

System.out.println("Default values are : ");

System.out.println("int : " + Test.a);

System.out.println("float : " + Test.b);

System.out.println("char : " + Test.c);

System.out.println("boolean : " + Test.d);

System.out.println("pointer : " + Test.e);

}

}

**Static user method:**

* Program execution automatically starts with main() method.
* Depends on application requirement, we can write any number of custom method.
* We must define static method with “static” keyword.

class Test

{

public static void main(String[] args)

{

System.out.println("Starts @ main");

}

static void fun()

{

System.out.println("fun...");

}

}

**Method is not taking input and not giving output:**

* Method executes only when we call.
* After execution, the control back to the calling area.
* We can call static method using “class name”

class Test

{

public static void main(String[] args)

{

System.out.println("Starts @ main");

Test.fun();

System.out.println("Ends @ main");

}

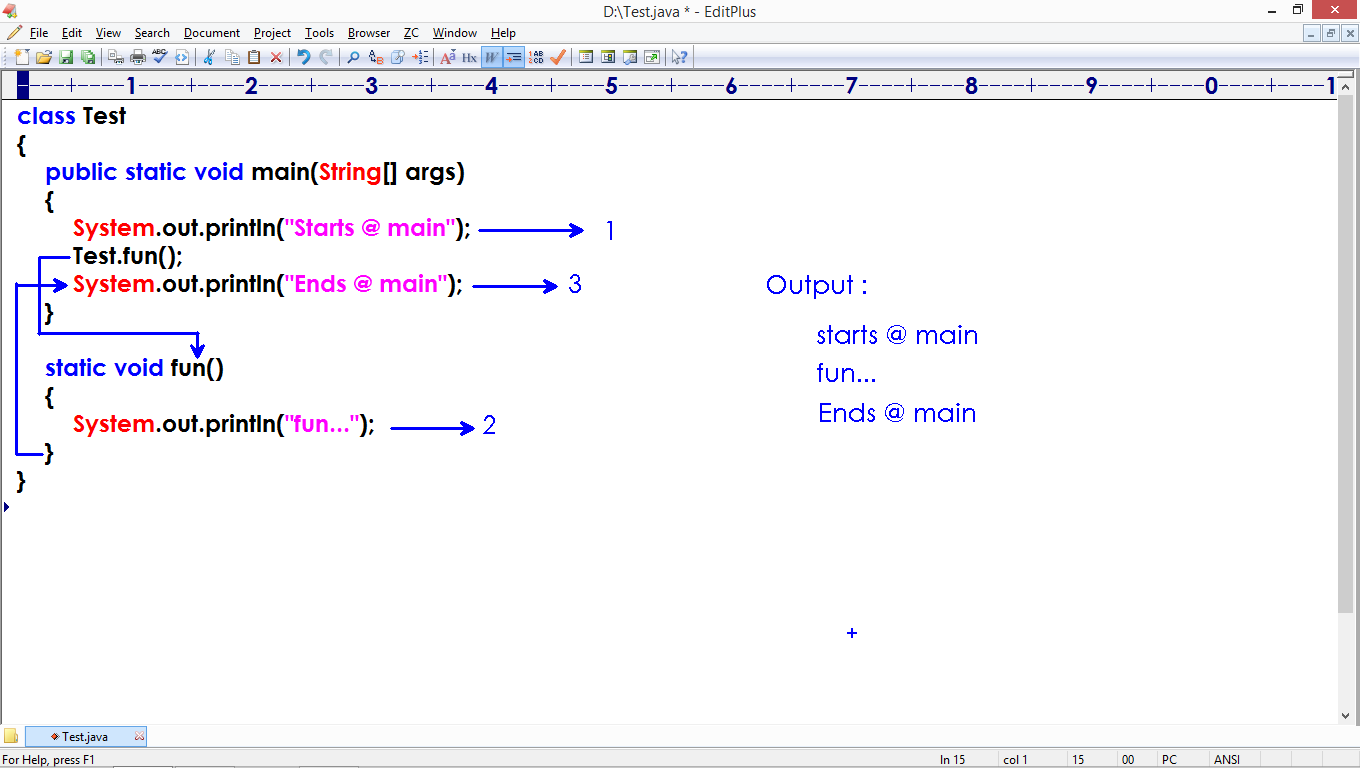
static void fun()

{

System.out.println("fun...");

}

}



class Pro

{

public static void main(String[] args)

{

System.out.println("main starts");

Pro.fun();

System.out.println("main ends");

}

static void fun()

{

System.out.println("fun");

}

static

{

System.out.println("Block starts");

Pro.fun();

System.out.println("Block ends");

}

}

* We can define any number of methods in a single class.
* Once the method has be defined, we can call any number of times.

class Test

{

static void m1()

{

System.out.println("m1...");

}

public static void main(String[] args)

{

System.out.println("Main");

Test.m2();

Test.m2();

Test.m1();

Test.m2();

System.out.println("End");

}

static void m2()

{

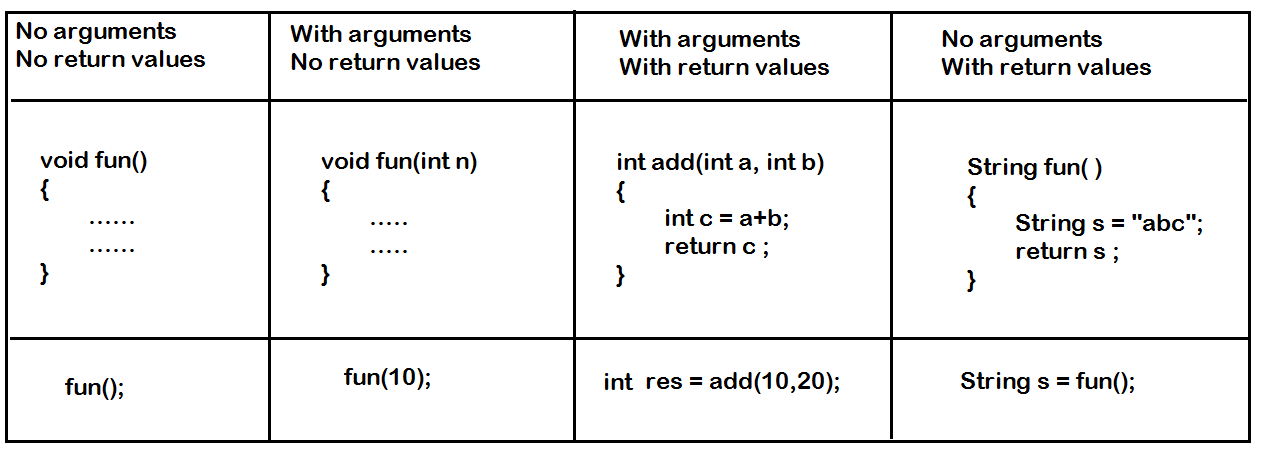
System.out.println("m2...");

}

}

**Methods classified into:**

* No arguments and No return values
* With arguments and No return values
* With arguments and With return values
* No arguments and With return values
* Recursion



**No arguments and No return values:**

class Pro

{

public static void main(String args[])

{

Pro.sayHi();

Pro.sayHi();

Pro.sayHi();

}

static void sayHi()

{

System.out.println("Hi every one");

}

}

**Method is taking input and no output:**

class Test

{

public static void main(String[] args)

{

Test.display("Hi");

Test.display("Hello");

Test.display("How r u");

}

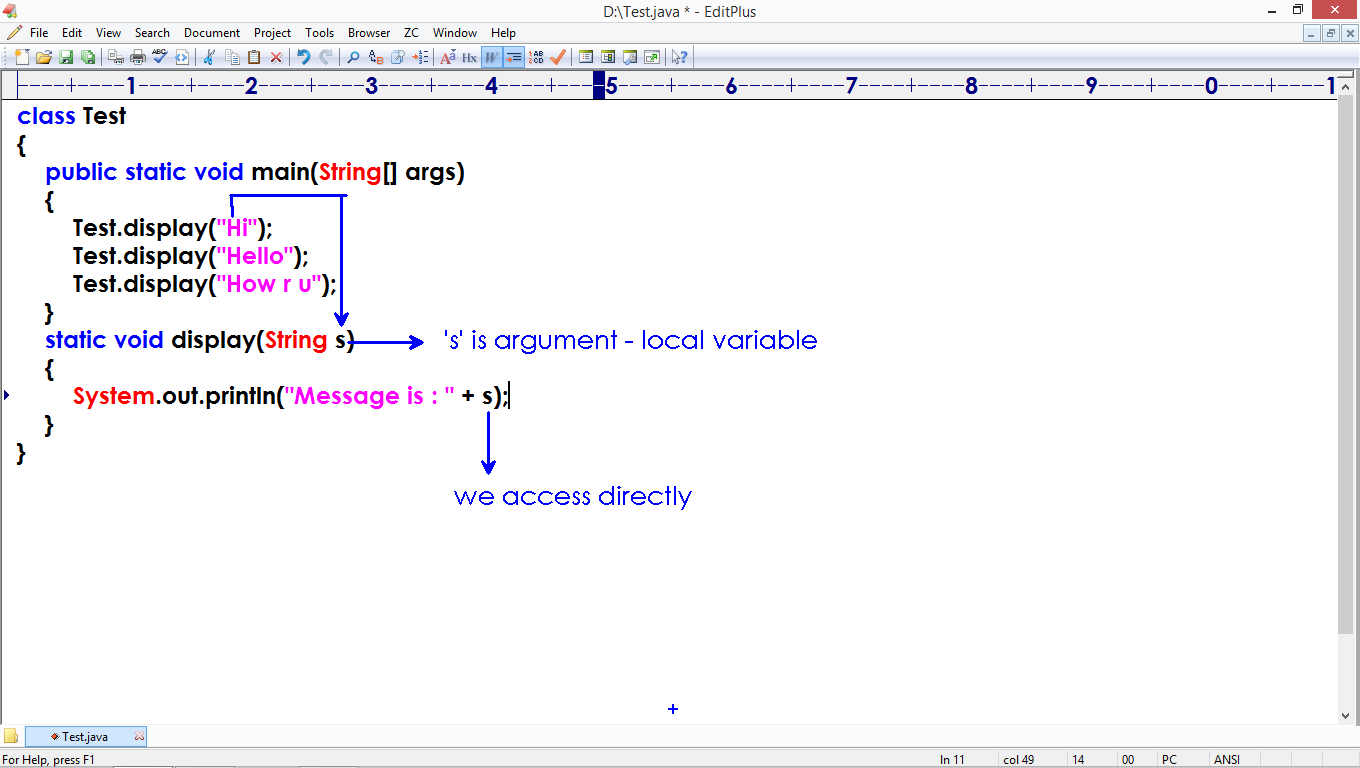
static void display(String s)

{

System.out.println("Message is : " + s);

}

}



class Test

{

public static void main(String[] args)

{

Test.display(23.45f);

Test.display(13);

Test.display(true);

}

static void display(int x)

{

System.out.println("int value is : " + x);

}

static void display(float x)

{

System.out.println("float value is : " + x);

}

static void display(boolean x)

{

System.out.println("boolean value is : " + x);

}

}

**Method is taking input and returning output:**

class Test

{

public static void main(String[] args)

{

int res = Test.add(10, 20);

System.out.println("Result is : " + res);

}

static int add(int a, int b)

{

int c = a+b;

return c ;

}

}

class Pro

{

public static void main(String args[])

{

int r1 = Pro.add(5,2);

System.out.println("sum is : " + r1);

int r2 = Pro.multiply(7,3);

System.out.println("Product is : " + r2);

}

static int add(int x, int y)

{

int z=x+y;

return z;

}

static int multiply(int a, int b)

{

return a\*b;

}

}

class Test

{

public static void main(String[] args)

{

int r1 = Test.add(10, 20);

System.out.println("r1 : " + r1);

float r2 = Test.add(10.5f, 20.3f);

System.out.println("r2 : " + r2);

double r3= Test.add(23.5, 34.6);

System.out.println("r3 : " + r3);

}

static int add(int a, int b)

{

int c = a+b;

return c ;

}

static float add(float a, float b)

{

float c = a+b;

return c ;

}

static double add(double a, double b)

{

double c = a+b;

return c ;

}

}

class Test

{

static int a = 10 ;

public static void main(String[] args)

{

int a = 20 ;

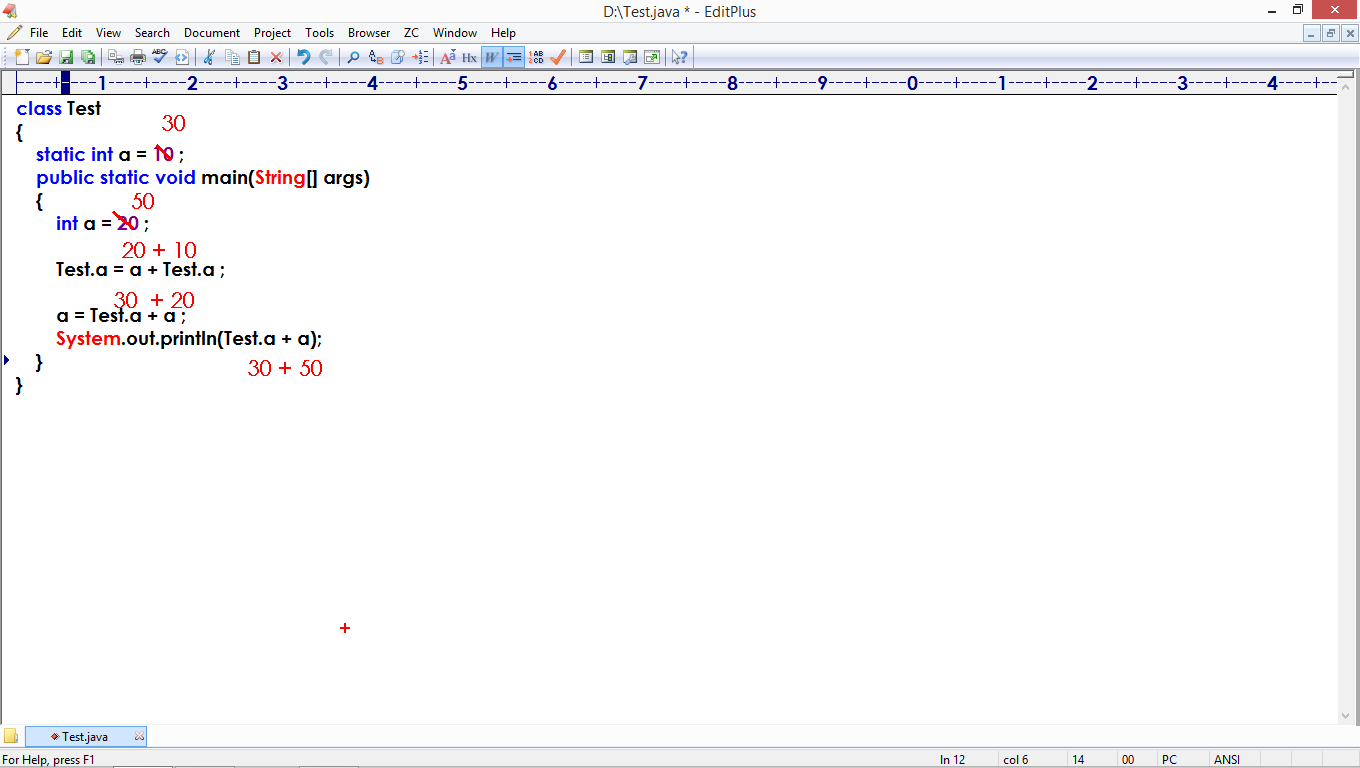
Test.a = a + Test.a ;

a = Test.a + a ;

System.out.println(Test.a + a);

}

}



class Test

{

static int a = 10 ;

static

{

int a = 20 ;

Test.a = Test.a + a ;

}

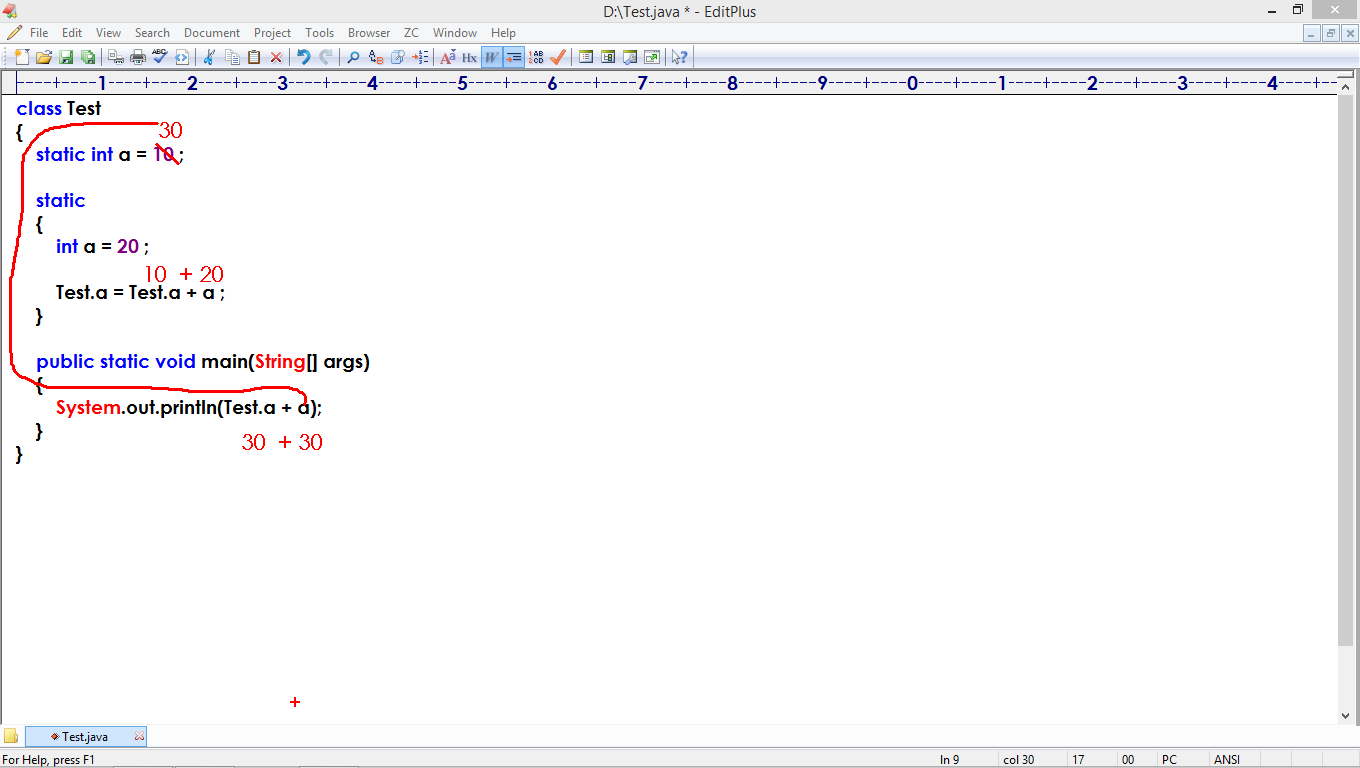
public static void main(String[] args)

{

System.out.println(Test.a + a);

}

}



class Test

{

static int a = 10 ;

static

{

int a = 20 ;

a = Test.a + a ;

}

public static void main(String[] args)

{

System.out.println(a + a);

}

static

{

Test.a = a + Test.a ;

}

}

class Pro

{

static int a=10;

public static void main(String[] args)

{

System.out.println("In main : " + Pro.a);

Pro.a = Pro.a + 25 ;

Pro.fun();

}

static

{

System.out.println("In block : " + Pro.a);

Pro.a = Pro.a + 15 ;

Pro.fun();

}

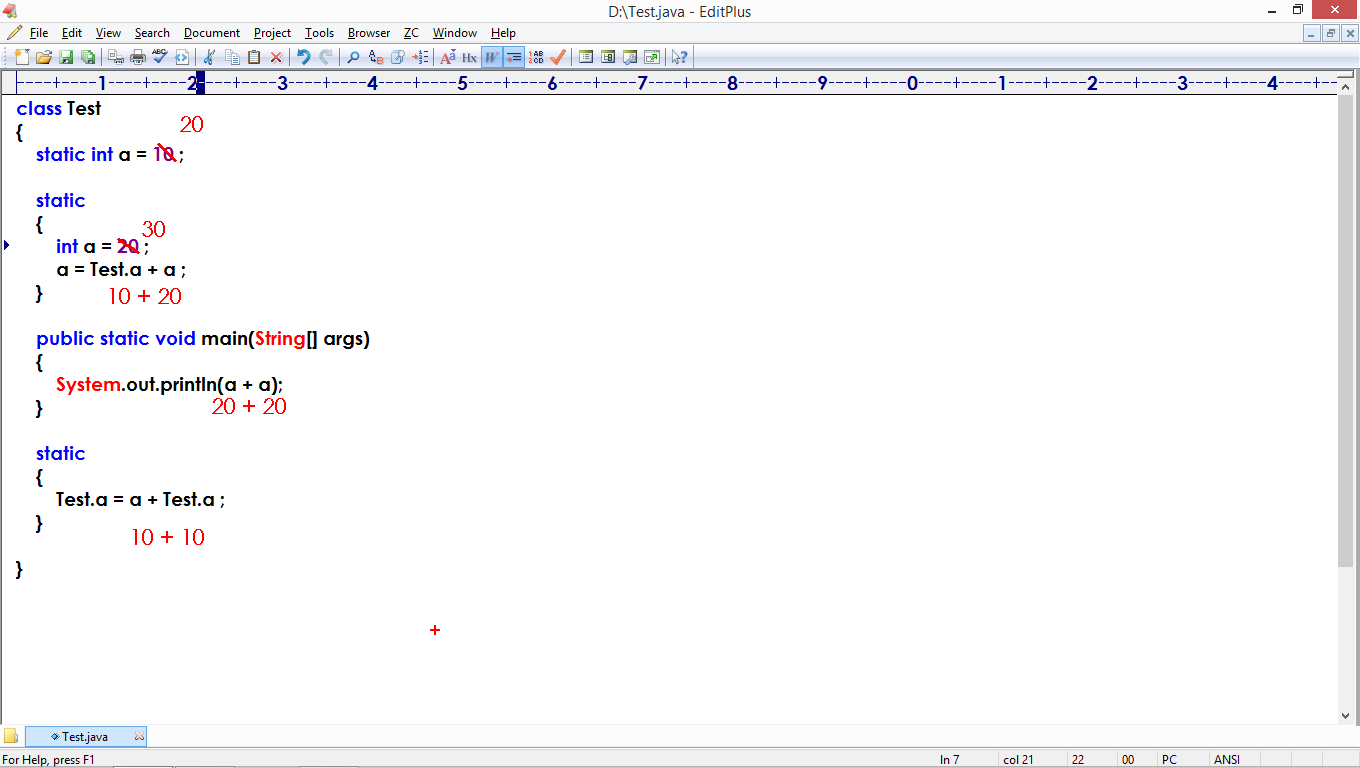
static void fun()

{

System.out.println("In fun : " + Pro.a);

}

}



class Test

{

static int a ;

static

{

int a = 10;

Test.a = a + Test.a;

}

public static void main(String[] args)

{

System.out.println(Test.a + a);

}

static

{

a = a + Test.a ;

}

}

class Test

{

static int a = 10;

public static void main(String[] args)

{

int res = Test.fun() + Test.a ;

System.out.println(res);

}

static int fun()

{

return Test.a + a ;

}

}

class Pro

{

static int a=10;

public static void main(String[] args)

{

System.out.println(Pro.a + Pro.fun() + Pro.a);

}

static int fun()

{

int a=5;

Pro.a = a + Pro.a ;

return a ;

}

}

class Pro

{

static int a=10;

public static void main(String[] args)

{

System.out.println(Pro.fun() + Pro.a + Pro.fun());

}

static int fun()

{

int a=5;

return a + Pro.a ;

}

}

class Pro

{

static int a=15;

public static void main(String[] args)

{

System.out.println(Pro.fun() + Pro.a + Pro.fun());

System.out.println(Pro.a + Pro.fun() + Pro.a);

}

static int fun()

{

int a=25;

Pro.a = Pro.a + a;

return a ;

}

}

class Test

{

static int a = 20;

public static void main(String[] args)

{

System.out.println(Test.fun() + Test.a + Test.fun());

}

static int fun()

{

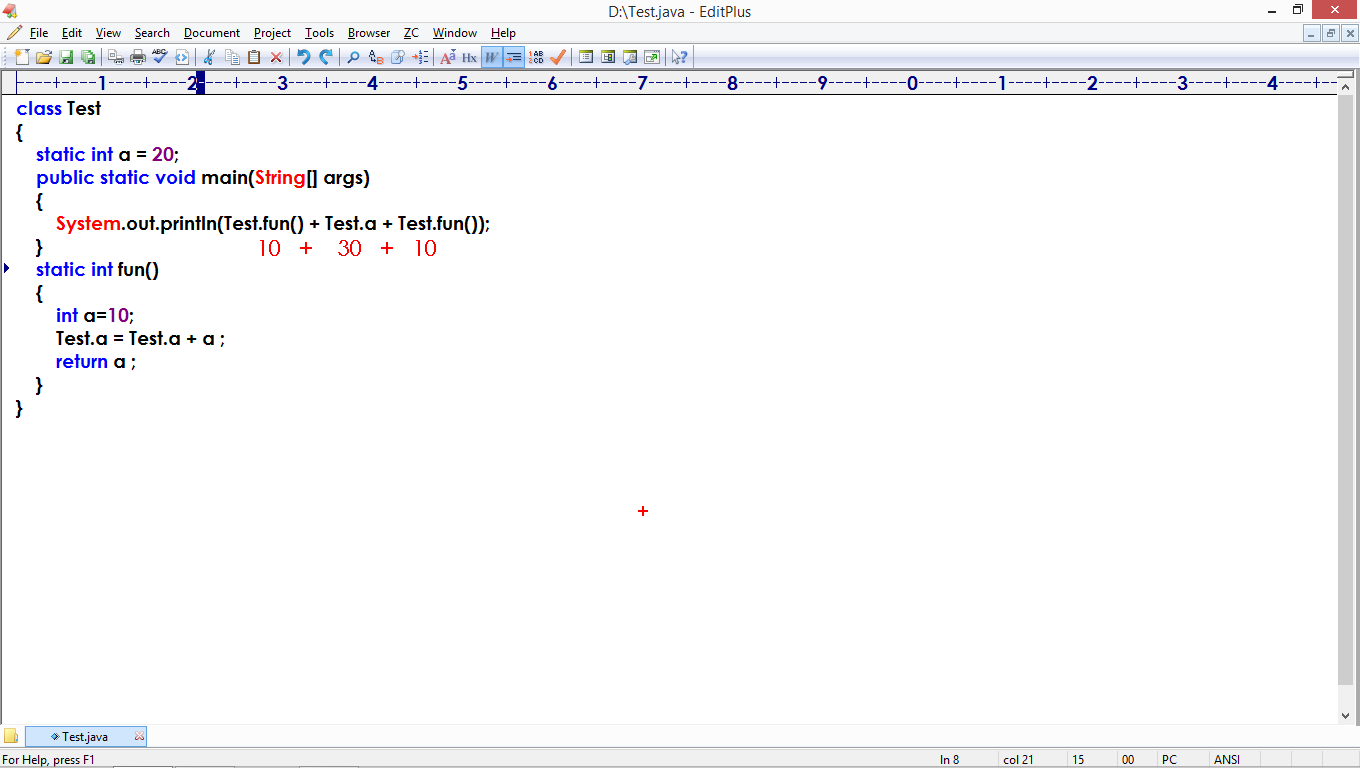
int a=10;

Test.a = Test.a + a ;

return a ;

}

}



class Test

{

static int a = 15;

public static void main(String[] args)

{

System.out.println(Test.fun() + Test.a + Test.fun());

System.out.println(Test.a + Test.fun() + Test.a);

}

static int fun()

{

int a=10;

Test.a = Test.a + a ;

return Test.a ;

}

}

class Pro

{

static int a=10;

static

{

System.out.println(Pro.fun() + Pro.a + Pro.fun());

a = Pro.a + a ;

}

public static void main(String[] args)

{

System.out.println(Pro.a + Pro.fun() + Pro.fun());

}

static int fun()

{

int a=5;

Pro.a = Pro.a + a;

return Pro.a ;

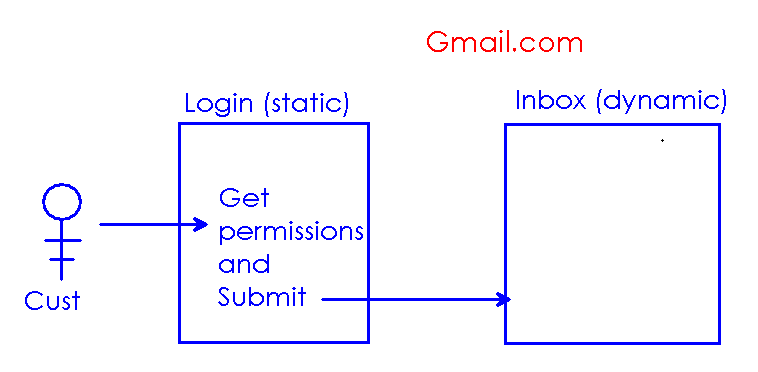
}

}

**Non static members**

Every java application has 2 contexts:

1. Static context : Free accessible area
2. Non static context : Restricted access.



**Static members are :**

1. Static main method
2. Static block
3. Static variable
4. Static user method

**Code:**

class Test

{

static int a = 10;

public static void main(String[] args)

{

int a=20;

System.out.println("In main : " + Test.a);

Test.a = Test.a + a ;

Test.fun();

}

static

{

System.out.println("In block : " + a);

Test.a = a + Test.a ;

}

static void fun()

{

System.out.println("In fun : " + a);

}

}

**Non static members:**

1. Non static block
2. Non static variable
3. Non static method
4. Constructor.

How can we access Non static members?

Ans : By creating object of that class.

**How to create object?**

Constructor:

* It is a special java method.
* We must define with class name
* No return type allowed.
* We must access constructor to create the object.

class Test

{

Test()

{

System.out.println("Constructor");

}

public static void main(String[] args)

{

new Test();

}

}



**Non static block:**

* A block of instructions with no identity.
* It is also called Instance block or Anonymous block.
* Non static block executes every time in object creation, before constructor call.

class Test

{

Test()

{

System.out.println("Constructor");

}

public static void main(String[] args)

{

new Test();

new Test();

new Test();

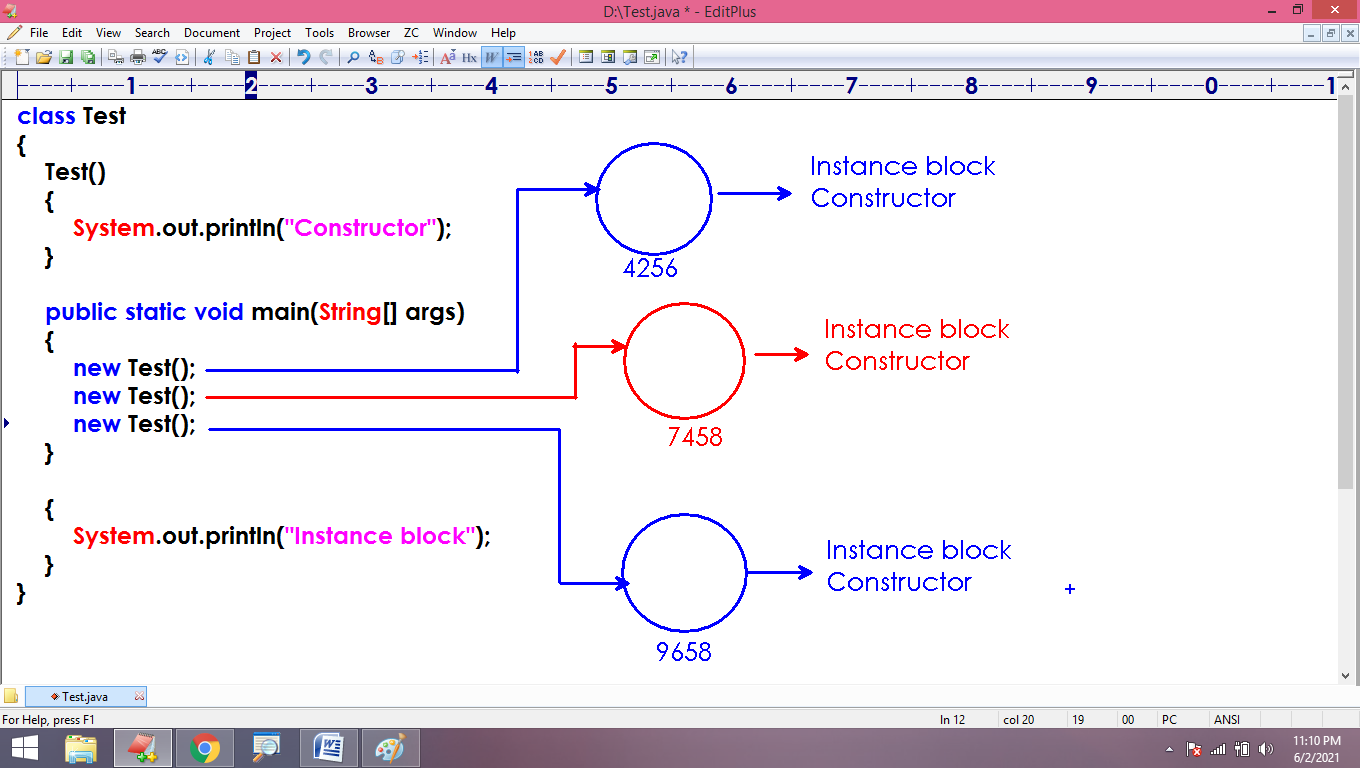
}

{

System.out.println("Instance block");

}

}



this:

* It is a keyword.
* It is a pre-defined variable.
* It holds object address.

class Test

{

Test()

{

System.out.println("Object created : " + this);

}

public static void main(String[] args)

{

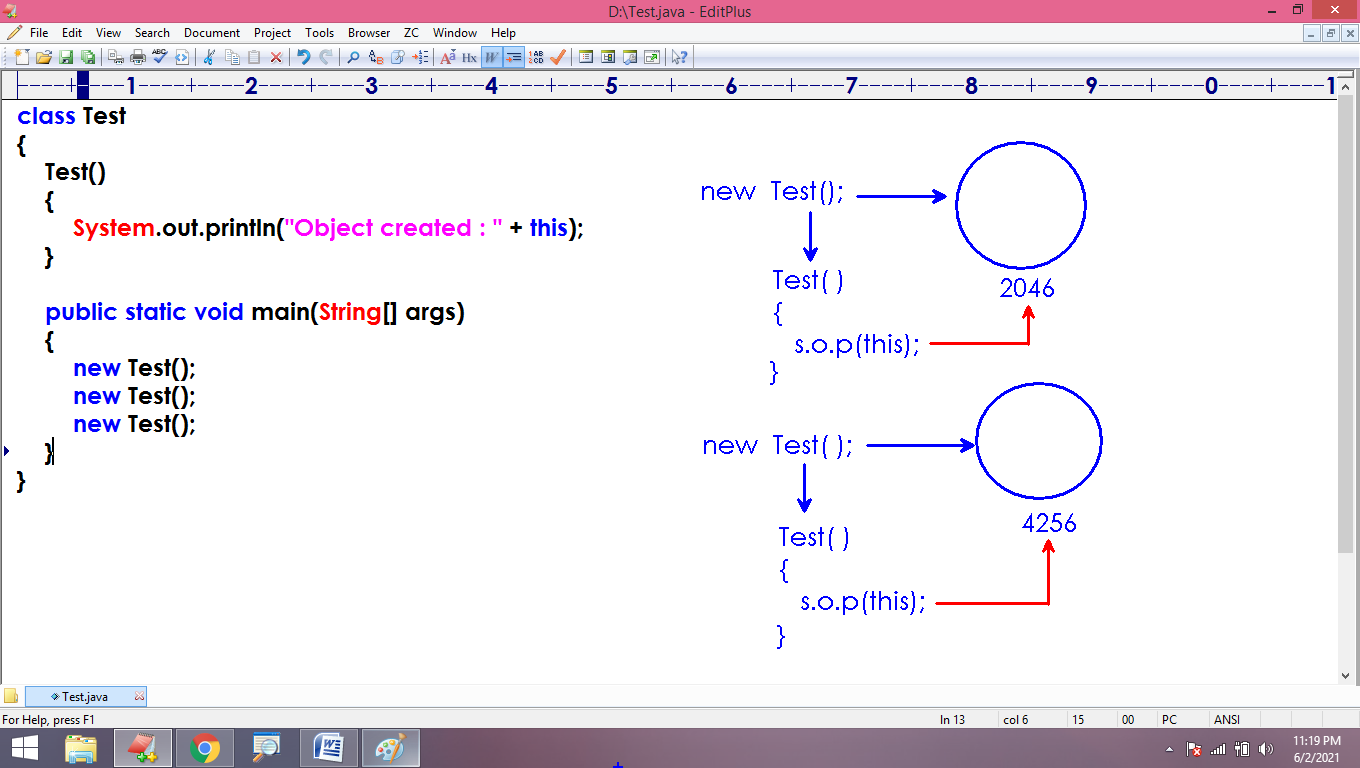
new Test();

new Test();

new Test();

}

}



Note: We must define ‘this’ keyword inside the non static area. Placing ‘this’ in static area results ‘error’

class Test

{

Test()

{

System.out.println(this); // allowed - it is non static area

}

public static void main(String[] args)

{

System.out.println(this); // not allowed - it is static area

}

static

{

System.out.println(this); // not allowed - it is static area

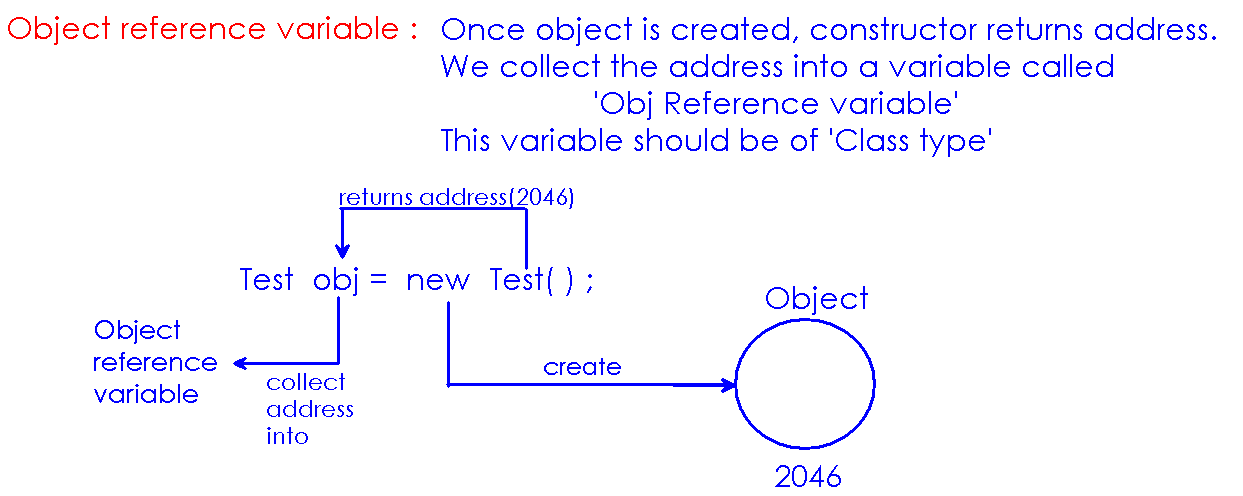
}

{

System.out.println(this); // allowed - it is non static area

}

}



class Test

{

Test()

{

System.out.println("Object created, address is : " + this);

}

public static void main(String[] args)

{

Test x = new Test();

System.out.println("x object address is : " + x);

}

}

/\*

Static member(variable/method) : Access using class name

Non static member(variable/method) : Access using object address.

\*/

class Test

{

public static void main(String[] args)

{

Test.m1();

Test obj = new Test();

obj.m2();

}

static void m1()

{

System.out.println("It is static method");

}

void m2()

{

System.out.println("It is non static method");

}

}

/\*

-> If we don't define any constructor, Compiler will add a default constructor.

-> Default constructor : No arguments and No logic

class\_name()

{

empty

}

\*/

class Test

{

/\* Test()

{

System.out.println("Constructor");

} \*/

public static void main(String[] args)

{

Test obj = new Test();

}

}

class Test

{

Test()

{

System.out.println("Object created....");

this.fun();

}

public static void main(String[] args)

{

Test obj = new Test();

obj.fun();

}

void fun()

{

System.out.println("It is non static fun...");

}

}

class Test

{

public static void main(String[] args)

{

Test obj = new Test();

obj.m1();

}

void m1()

{

System.out.println("m1");

this.m2();

}

void m2()

{

System.out.println("m2");

Test.m3();

}

static void m3()

{

System.out.println("m3");

}

}

Non static variable:

* Defining a variable inside the class and outside to all blocks and methods.
* We define non static variable without static keyword.

/\*

Java supports :

1. static variable

2. non static variable

3. local variable

\*/

class Test

{

static int a ;

int b ;

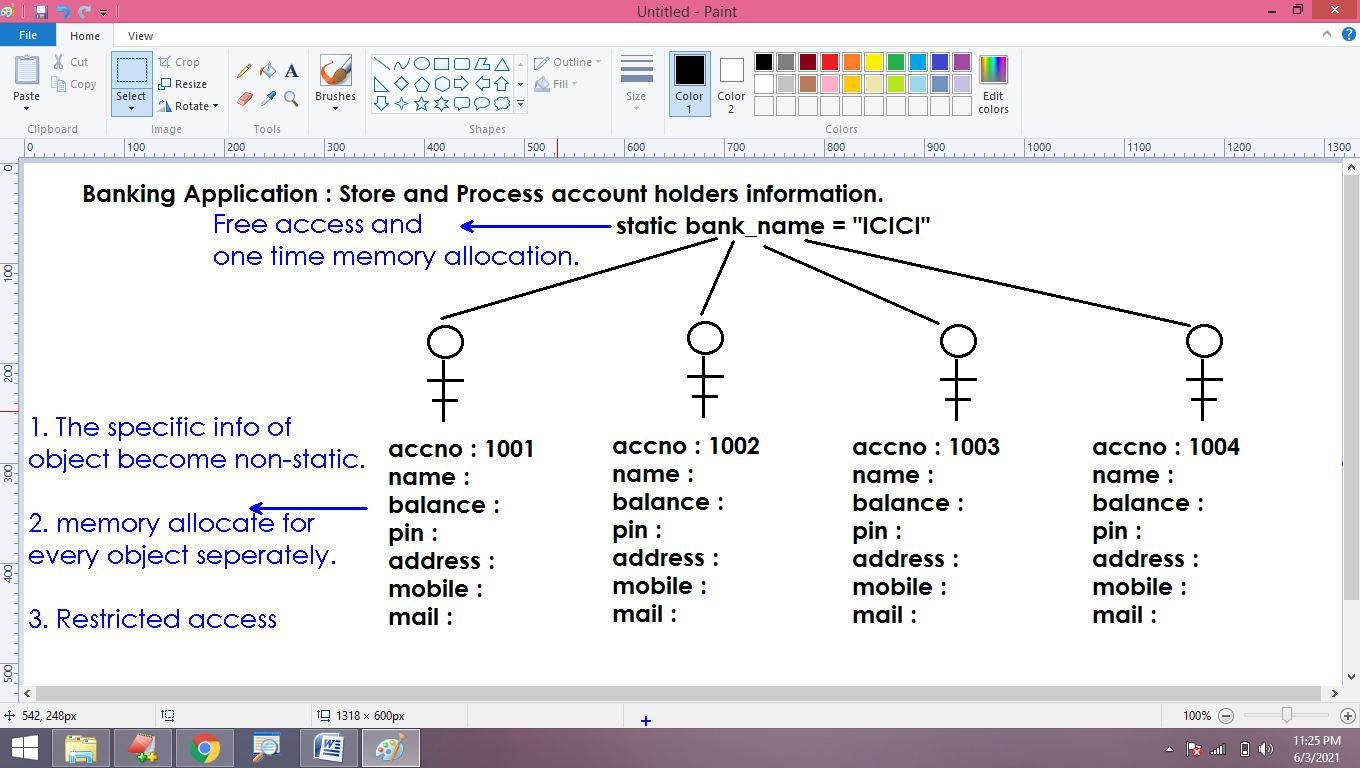
public static void main(String args[ ])

{

int c ;

}

}



class Test

{

int a, b;

Test()

{

this.a = 10 ;

this.b = 20 ;

}

public static void main(String[] args)

{

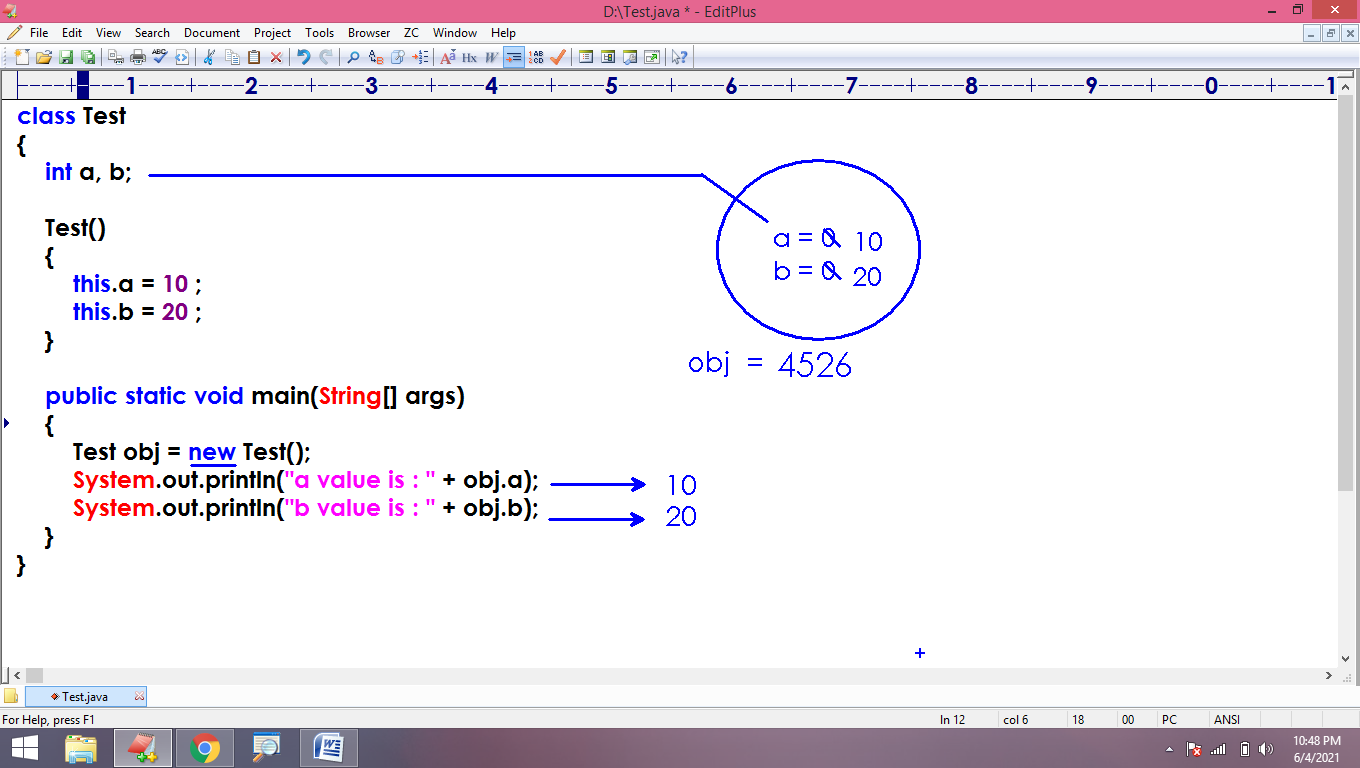
Test obj = new Test();

System.out.println("a value is : " + obj.a);

System.out.println("b value is : " + obj.b);

}

}



Non static variables get memory inside the object and initializes with default values automatically.

class Test

{

int a ;

float b ;

boolean c;

public static void main(String[] args)

{

Test obj = new Test();

System.out.println("a value is : " + obj.a);

System.out.println("b value is : " + obj.b);

System.out.println("c value is : " + obj.c);

}

}

class Test

{

int a, b;

Test()

{

this.a = 10 ;

this.b = 20 ;

}

public static void main(String[] args)

{

Test t1 = new Test();

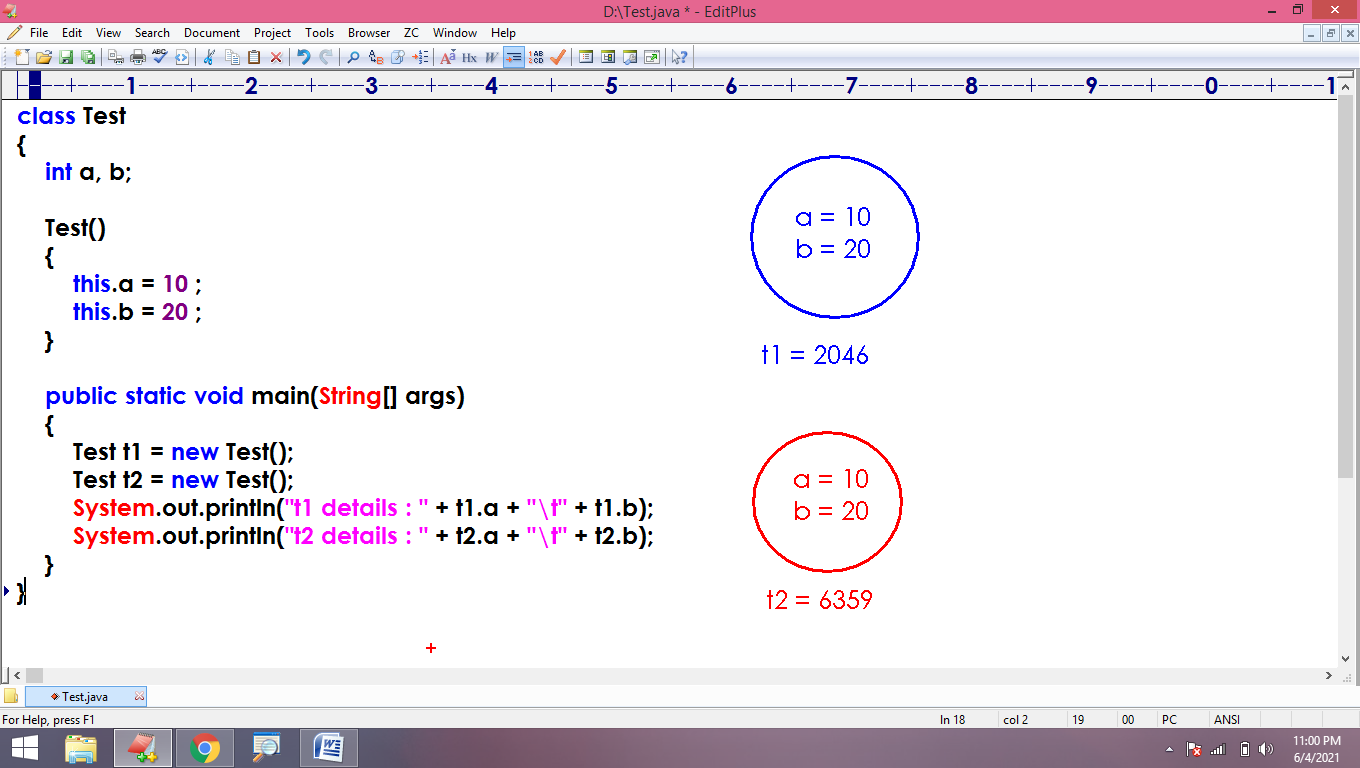
Test t2 = new Test();

System.out.println("t1 details : " + t1.a + "\t" + t1.b);

System.out.println("t2 details : " + t2.a + "\t" + t2.b);

}

}



Arguments constructor:

* A constructor is taking input.
* Constructor is used to initialize non static variables as soon as object has created.
* Using arguments constructor, we can set different values to different objects.

class Test

{

int a, b;

Test(int x, int y)

{

this.a = x;

this.b = y;

}

public static void main(String[] args)

{

Test t1 = new Test(10,20);

Test t2 = new Test(30,40);

System.out.println("t1 details : " + t1.a + "\t" + t1.b);

System.out.println("t2 details : " + t2.a + "\t" + t2.b);

}

}

* We can define both non static and local variables with the same name.
* Arguments – local variables
* Local variables – access directly
* Non static variables – access using object address.

class Test

{

int a, b;

Test(int a, int b)

{

this.a = a;

this.b = b;

}

public static void main(String[] args)

{

Test t1 = new Test(10,20);

Test t2 = new Test(30,40);

System.out.println("t1 details : " + t1.a + "\t" + t1.b);

System.out.println("t2 details : " + t2.a + "\t" + t2.b);

}

}

class Emp

{

int id;

String name;

double salary;

Emp(int id, String name, double salary)

{

this.id = id ;

this.name = name;

this.salary = salary;

}

}

class Test

{

public static void main(String[] args)

{

Emp e1 = new Emp(101, "Amar", 35000);

Emp e2 = new Emp(102, "annie", 45000);

Emp e3 = new Emp(103, "ananya", 55000);

System.out.println("e1 details are : " + e1.id + " \t " + e1.name + " \t " + e1.salary);

}

}

Class type variable:

* It is called object reference variable.
* It holds address(memory location) of object.
* Pointer type variables hold memory addresses.
* All class type variables are pointer type.
* The default value of pointer type variable is “null”

class Test

{

static Test x ;

public static void main(String[] args)

{

System.out.println("x val : " + Test.x);

}

}

class Test

{

static Test x ;

static String y ;

static System z ;

public static void main(String[] args)

{

int x=10;

float y=23.45f;

boolean z=false;

System.out.println("Local : \n" + x + "\n" + y + "\n " + z);

System.out.println("Static : \n" + Test.x + "\n" + Test.y + "\n" + Test.z);

}

}

class Test

{

static Test x ;

static

{

System.out.println("Address in Block : " + x);

x = new Test();

}

public static void main(String[] args)

{

System.out.println("Address in main : " + x);

}

}

class Test

{

static Test x ;

static

{

int x = 10 ;

System.out.println("Address in block : " + Test.x);

Test.x = new Test();

}

public static void main(String[] args)

{

System.out.println("Address in main : " + Test.x);

}

}

class Test

{

Test()

{

this.fun();

}

public static void main(String[] args)

{

Test obj = new Test();

obj.fun();

}

void fun()

{

System.out.println("fun method...");

}

}

class Test

{

int x, y;

Test(int a, int b)

{

this.x = a ;

this.y = b ;

}

public static void main(String[] args)

{

Test obj = new Test(10, 20);

obj.display();

}

void display()

{

System.out.println("x value : " + this.x);

System.out.println("y value : " + this.y);

}

}

class Test

{

int x, y;

Test(int a, int b)

{

this.x = a ;

this.y = b ;

}

public static void main(String[] args)

{

Test obj = new Test(10, 20);

obj.display();

}

void display()

{

int x=30, y=40;

System.out.println("Local : " + x + " , " + y);

System.out.println("x value : " + this.x);

System.out.println("y value : " + this.y);

}

}

class Test

{

static Test x = new Test(); // object created and address stored in 'x'

static

{

int x = 10;

System.out.println("Address in block : " + Test.x);

}

public static void main(String[] args)

{

int x = 20 ;

System.out.println("Address in main : " + Test.x);

}

}

class Test

{

static Test x = new Test();

static

{

int x = 10;

// access fun()

}

public static void main(String[] args)

{

int x = 20 ;

// access fun()

}

void fun()

{

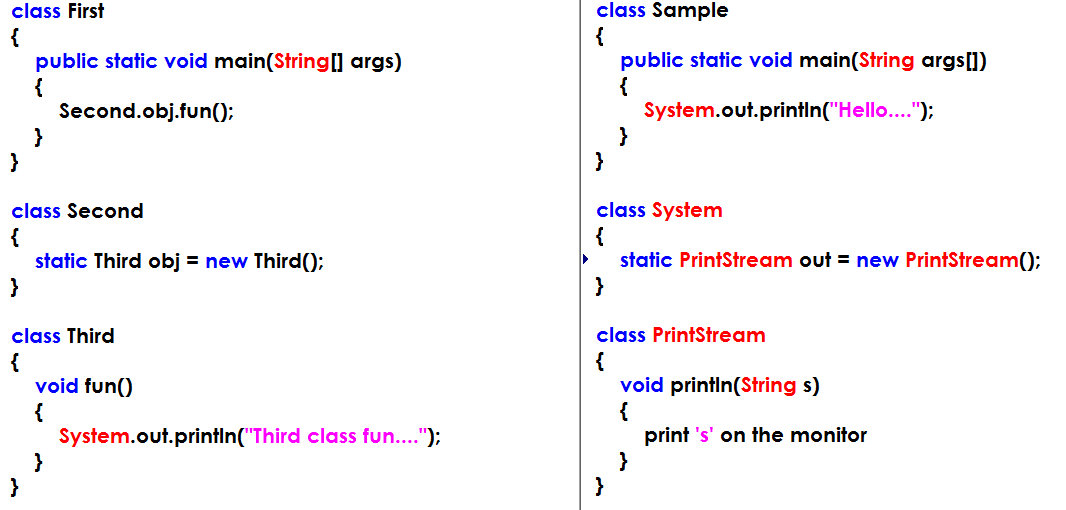
System.out.println("fun method....");

}

}

**System.out.println():**

* println() is a non static method belongs to PrintStream class.
* Object of PrintStream class is available in System class as “static variable”
* To call println(), we use object present in System class.





class Student

{

public static void main(String[] args)

{

Faculty.permission.copyMaterials();

}

}

class Faculty

{

static Laptop permission = new Laptop();

}

class Laptop

{

void copyMaterials()

{

System.out.println("copying materials...");

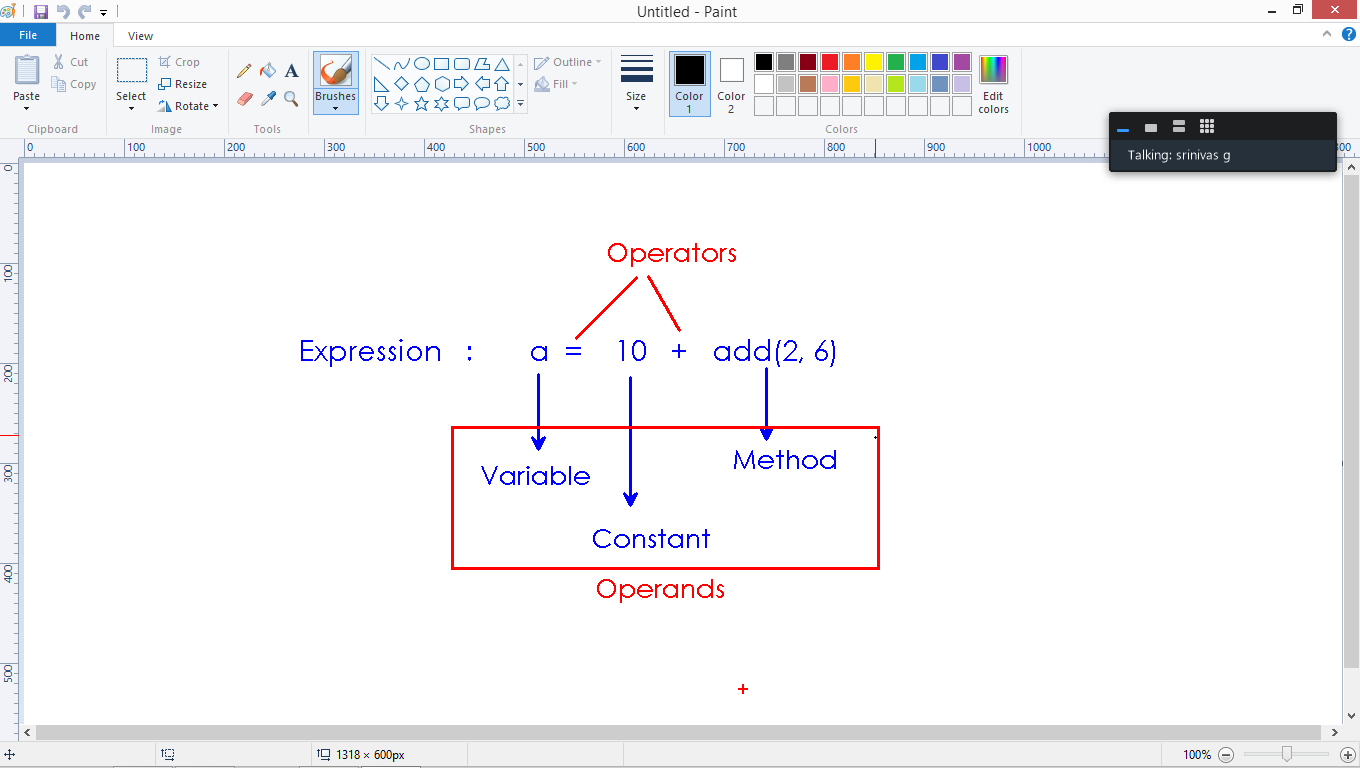
}

}

**Logical programming**

(Operators & Control Statements)

Operator: It is a symbol that performs an operation on one or more operands



1. Arithmetic operators : Performs all arithmetic operations
   1. Operators : +, - , \* , / , %
2. ‘/’ operator returns quotient
3. ‘%’ operator returns remainder

class Test

{

public static void main(String[] args)

{

int a=5 , b=3 ;

System.out.println("Arithmetic opertors : ");

System.out.println(a+ "+" +b+ "=" + (a+b));

System.out.println(a+ "-" +b+ "=" + (a-b));

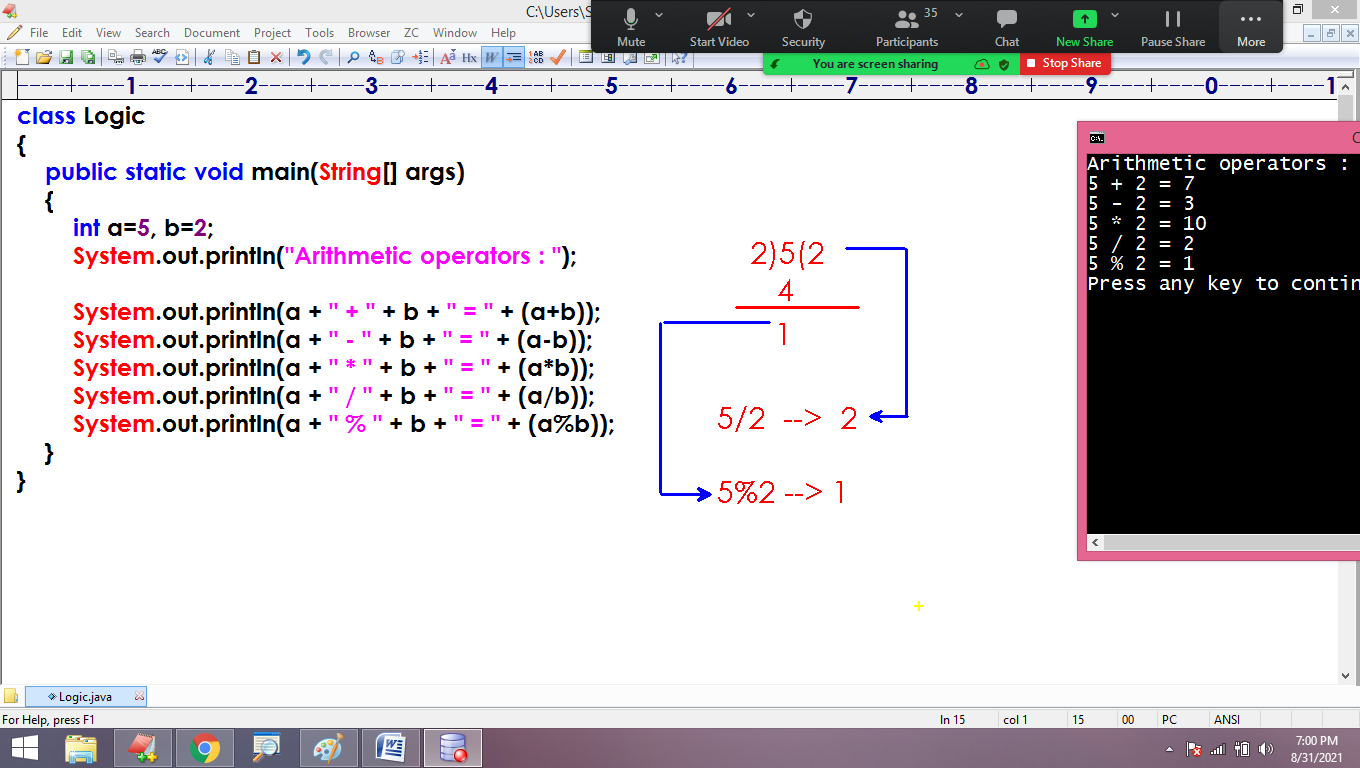
System.out.println(a+ "\*" +b+ "=" + (a\*b));

System.out.println(a+ "/" +b+ "=" + (a/b));

System.out.println(a+ "%" +b+ "=" + (a%b));

}

}



**Relational operators:**

* These operators return a boolean value by validating the relation between the operands
* Operators >, <, >=, <=, ==, !=

class Test

{

public static void main(String[] args)

{

int a=5 , b=3 ;

System.out.println("Relational opertors : ");

System.out.println(a+ ">" +b+ "=" + (a>b));

System.out.println(a+ "<" +b+ "=" + (a<b));

System.out.println(a+ ">=" +b+ "=" + (a>=b));

System.out.println(a+ "<=" +b+ "=" + (a<=b));

System.out.println(a+ "==" +b+ "=" + (a==b));

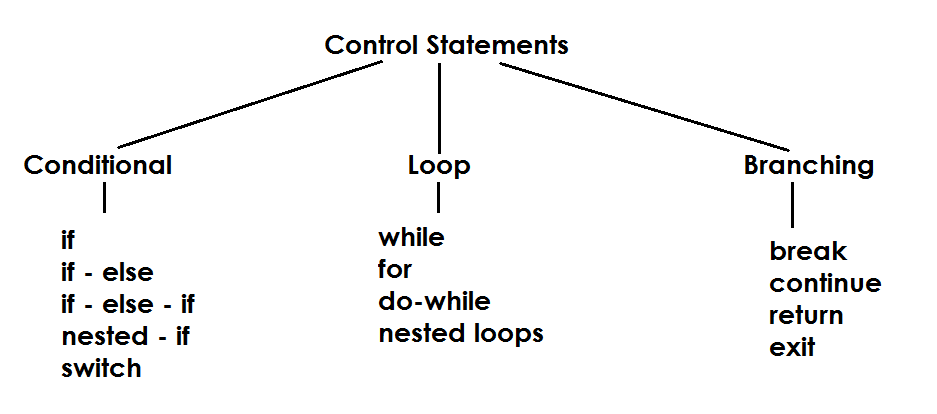
System.out.println(a+ "!=" +b+ "=" + (a!=b));

}

}

**Control Statements:**

* A statement is a line of code.
* Statements execute sequentially in the method from top to bottom
* We can control execution process using control statements



If-block:

* ‘if’ is a keyword
* It is used to execute a block only if the condition is valid.
* Condition always a boolean type.

class Test

{

public static void main(String[] args)

{

if(3!=5)

{

System.out.println("Hi....");

}

if(3>5)

{

System.out.println("Hello....");

}

if(true)

{

System.out.println("How r u....");

}

if(false)

{

System.out.println("Bye....");

}

}

}

class Logic

{

public static void main(String[] args)

{

if(3 != 5)

{

System.out.println("One");

System.out.println("Two");

}

if(true)

{

System.out.println("Three");

System.out.println("Four");

}

if(false)

{

System.out.println("Five");

System.out.println("Six");

}

}

}

class Logic

{

public static void main(String[] args)

{

if(false)

{

System.out.println("One");

System.out.println("Two");

System.out.println("Three");

}

}

}

* We need to block only multiple statements.
* If we don’t specify braces({ }), only the first statement consider as if-block statement.
* The if-condition is applicable to first statement only.

class Logic

{

public static void main(String[] args)

{

if(false)

System.out.println("One");

System.out.println("Two");

System.out.println("Three");

}

}

If-else:

* ‘else’ is a keyword.
* ‘else’ block is used to execute a set of statements when if-block condition fails.

class Test

{

public static void main(String[] args)

{

if(false)

System.out.println("Hi....");

else

System.out.println("Hello....");

}

}

Check biggest of 2 numbers using static method:

class Logic

{

public static void main(String[] args)

{

int a=10, b=20;

if(a>b)

System.out.println("a is big");

else

System.out.println("b is big");

}

}

class Logic

{

public static void main(String[] args)

{

int a=10, b=20;

Logic.big(a, b);

}

static void big(int a, int b)

{

if(a>b)

System.out.println("a is big");

else

System.out.println("b is big");

}

}

Check the input number is Even or Not

class Logic

{

public static void main(String[] args)

{

Logic obj = new Logic();

obj.even(6);

obj.even(13);

obj.even(17);

obj.even(22);

}

void even(int n)

{

if(n%2==0)

System.out.println(n + " is even");

else

System.out.println(n + " is not even");

}

}

class Logic

{

public static void main(String[] args)

{

if(true)

System.out.println("Hi");

else

System.out.println("Hello");

System.out.println("Bye");

}

}

**Logical operators:**

* Operators are
  + Logical – AND (&&)
  + Logical – OR (||)
  + Logical – NOT (!)
* These operators validate more than one expression and returns a boolean value

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Expr1** | **Expr2** | **&&** | **||** | **! Expr1** |
| T | T | T | T | F |
| T | F | F | T | F |
| F | T | F | T | T |
| F | F | F | F | T |

class Test

{

public static void main(String[] args)

{

System.out.println("Logical Operators");

System.out.println("5>3 && 5!=3 : " + (5>3 && 5!=3));

System.out.println("3<4 || 3==4 : " + (3<4 || 3==4));

System.out.println("true || false : " + (true || false));

System.out.println("!false : " + (!false));

}

}

class Logic

{

public static void main(String[] args)

{

System.out.println("Logical Operators");

System.out.println("5<3 && 5!=3 : " + (5<3 && 5!=3));

System.out.println("3>4 || 3==4 : " + (3>4 || 3==4));

System.out.println("false || (!true) : " + (false || !true));

System.out.println("!true : " + (!true));

}

}

**If-else-if:**

* It is used to defined more than one if block one is dependent to another.
* All the if-conditional evaluate from top to bottom and execute the block on valid condition.
* Only one block gets execute among all we define.

class Test

{

public static void main(String[] args)

{

if(false)

System.out.println("One");

else if(true)

System.out.println("Two");

else

System.out.println("Three");

}

}

class Test

{

public static void main(String[] args)

{

if(false)

System.out.println("One");

else if(!true)

System.out.println("Two");

else

System.out.println("Three");

}

}

Biggest of 3 numbers:

class Test

{

public static void main(String[] args)

{

Test.big(10, 20, 30);

}

static void big(int a, int b, int c)

{

if(a>b && a>c)

System.out.println("a is big");

else if(b>c)

System.out.println("b is big");

else

System.out.println("c is big");

}

}

**Nested – if block:**

* Defining if-block inside another if-block
* Nested if condition evaluates only when outer condition is valid.

class Test

{

public static void main(String[] args)

{

if(true)

{

System.out.println("Outer if block logic");

if(true)

{

System.out.println("Nested if block logic");

}

}

}

}

class Test

{

public static void main(String[] args)

{

if(false)

{

System.out.println("Outer if block logic");

if(true)

{

System.out.println("Nested if block logic");

}

}

}

}

/\* Number divisibility test with 3 and 5

n = 15 : Divisible by both 3 and 5

n = 10 : Divisible by 5 not 3

n = 9 : Divisible by 3 not 5

n = 7 : Not divisible by 3 or 5

\*/

class Test

{

public static void main(String[] args)

{

int n=10;

if(n%3==0)

{

if(n%5==0)

System.out.println("Divisible by both 3 and 5");

else

System.out.println("Divisible by 3 not 5");

}

else

{

if(n%5==0)

System.out.println("Divisible by 5 not 3");

else

System.out.println("Not divisible by 3 or 5");

}

}

}

/\*

Modify operators:

-> Operators are ++ , --

-> Operators are used to increase or decrease value of variable by 1

int a = 10;

++a --> a = a+1

-> Pre-increment (++a) will increase the value before substitution

\*/

class Test

{

public static void main(String[] args)

{

int x=5 ;

System.out.println("++x value is : " + ++x);

System.out.println("x value is : " + x);

}

}

/\*

-> a++ is called post increment operator

-> The value increases after substitution.

\*/

class Test

{

public static void main(String[] args)

{

int x=5 ;

System.out.println("x++ value is : " + x++);

System.out.println("x value is : " + x);

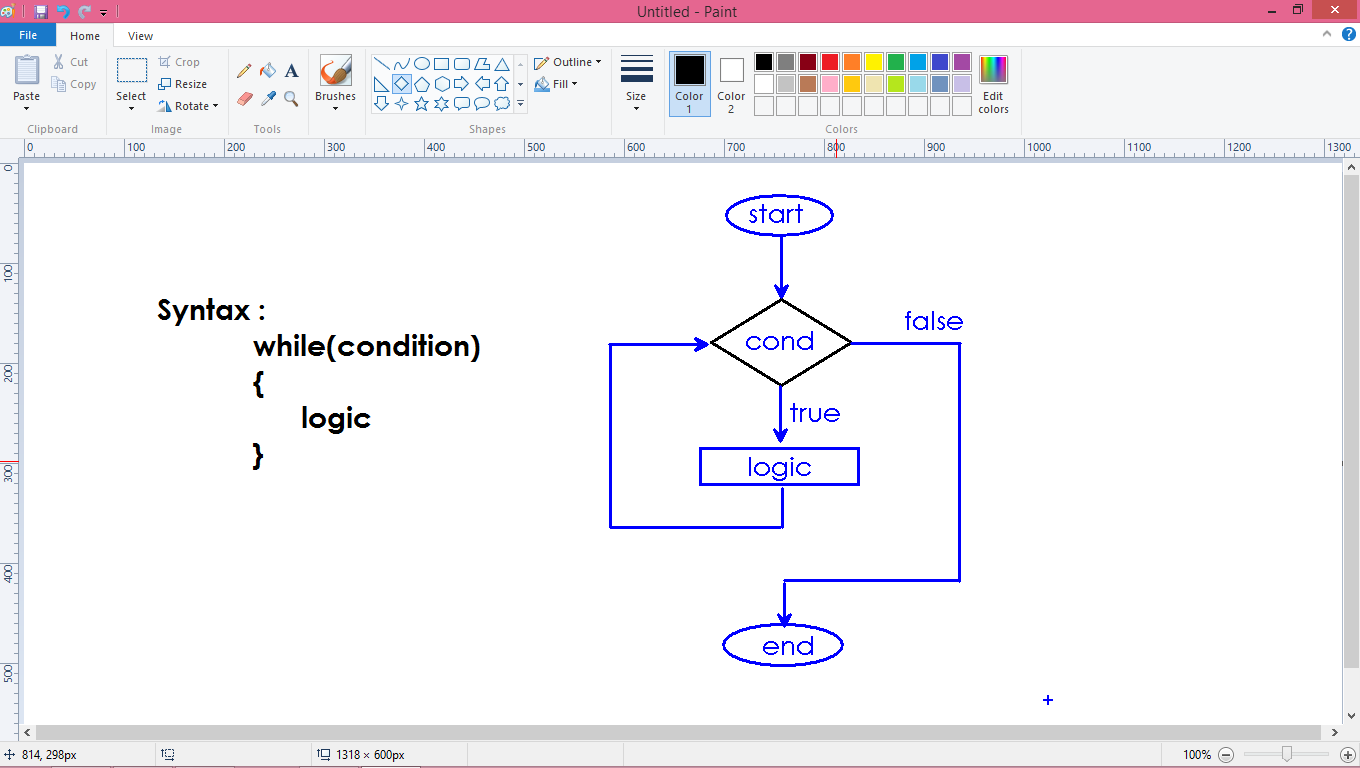
}

}

**Loop Control Statements:**

* Loops are used to execute a block of instructions repeatedly as long as the condition is valid.
* Java supports
  + While loop
  + For loop
  + Do-while loop

**While loop:**



class Test

{

public static void main(String[] args)

{

while(true)

{

System.out.println("Loop...");

}

}

}

class Test

{

public static void main(String[] args)

{

int i=1;

while(i<=10)

{

System.out.println("i value : " + i);

++i ;

}

}

}

**Display even numbers from 1 to 10:**

class Test

{

public static void main(String[] args)

{

int i=1;

while(i<=10)

{

if(i%2==0)

{

System.out.println("i value : " + i);

}

++i ;

}

}

}

Nested while loop:

* Defining while loop inside another while loop.
* Loop executes continuously as long as the condition is valid.

class Test

{

public static void main(String[] args)

{

while(true)

{

System.out.println("Outer logic");

while(true)

{

System.out.println("Nested logic");

}

}

}

}

class Test

{

public static void main(String[] args)

{

int x=1;

while(x<=5)

{

System.out.println("Outer x val : " + x);

++x ;

int y=1;

while(y<=5)

{

System.out.println("Nested y val : " + y);

++y;

}

}

}

}

For loop:

* The simple way of repeating a block based on condition.
* We can place initialization, condition and modify statement in a single line.

class Test

{

public static void main(String[] args)

{

for(int i=1 ; i<=10 ; i++)

{

System.out.println("i value : " + i);

}

}

}

**Even numbers:**

class Test

{

public static void main(String[] args)

{

for(int i=1 ; i<=10 ; i++)

{

if(i%2==0)

System.out.println("i value : " + i);

}

}

}

/\*

Scanner :

-> It is pre-defined library class.

-> It is used to read the data from end user.

\*/

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter 2 numbers : ");

int a = scan.nextInt();

int b = scan.nextInt();

int c = a+b;

System.out.println("Sum is : " + c);

}

}

**Print Multiplication table:**

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter table number : ");

int n = scan.nextInt();

for(int i=1 ; i<=10 ; i++)

{

System.out.println(n + "x" + i + "=" + (n\*i));

}

}

}

**Sum of First N numbers:**

// n=5 --> 1+2+3+4+5 = 15

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int sum=0;

for(int i=1 ; i<=n ; i++)

{

sum = sum + i;

}

System.out.println("Sum of First " + n + " numbers is : " + sum);

}

}

**Factorial of a given number:**

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int fact=1;

for(int i=n ; i>0 ; i--)

{

fact = fact\*i;

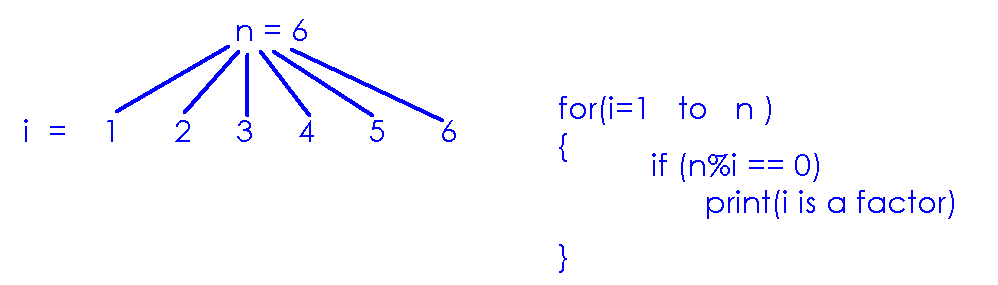
}

System.out.println("Factorial is : " + fact);

}

}

**Find factors to given number:**



import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

for(int i=1 ; i<=n ; i++)

{

if(n%i==0)

System.out.println(i + " is a factor");

}

}

}

Define a static method that returns number of factors of input number:

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int count = Test.factors\_count(n);

System.out.println("Factors count is : " + count);

}

static int factors\_count(int n)

{

int count=0;

for(int i=1 ; i<=n ; i++)

{

if(n%i == 0)

++count ;

}

return count;

}

}

WAP to check the input number is Prime or Not?

Prime Number: The number which is having 2 factors.

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

Test.prime(n);

}

static void prime(int n)

{

int count=0;

for (int i=1 ; i<=n ; i++)

{

if(n%i==0)

++count;

}

if(count==2)

System.out.println("It is prime number");

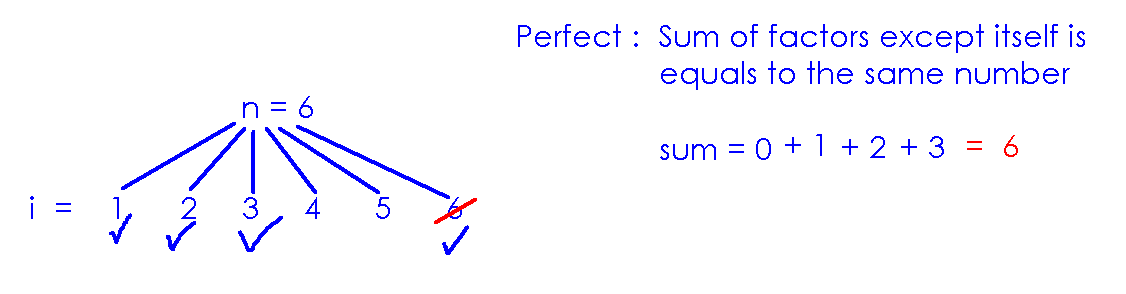
else

System.out.println("It is not prime number");

}

}

Perfect Number program:



import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

Test.perfect(n);

}

static void perfect(int n)

{

int sum=0;

for (int i=1 ; i<n ; i++)

{

if(n%i==0)

sum=sum+i;

}

if(n==sum)

System.out.println("It is perfect number");

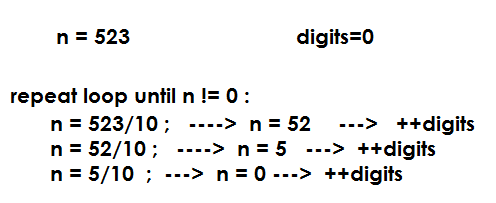
else

System.out.println("It is not perfect number");

}

}

WAP to count digits in the given number:



import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int digits=0;

while(n != 0)

{

n = n/10 ;

++digits ;

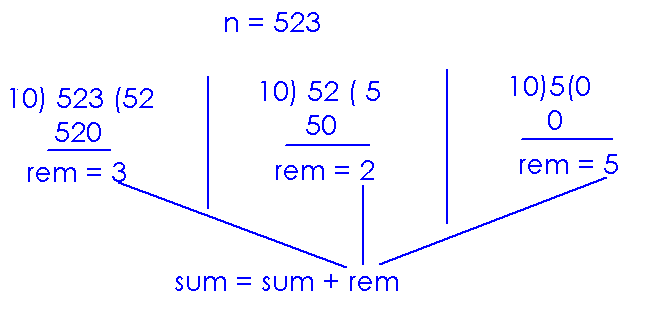
}

System.out.println("Digits count is : " + digits);

}

}

WAP to find the sum of all digits:



import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int sum=0, rem;

while(n != 0)

{

rem = n%10;

sum = sum + rem ;

n = n/10 ;

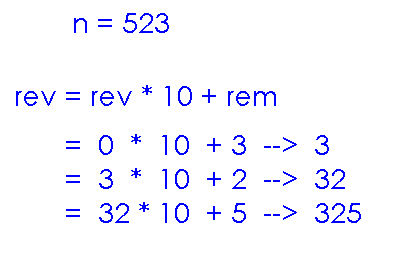
}

System.out.println("Sum is : " + sum);

}

}

Reverse number program:



import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int rev=0, rem;

while(n != 0)

{

rem = n%10;

rev = rev \* 10 + rem ;

n = n/10 ;

}

System.out.println("Rererse number is : " + rev);

}

}

Palindrome number: If the reverse number is as same as the give number is called palindrome. For example 5225

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value : ");

int n = scan.nextInt();

int rev=0, rem, temp=n;

while(n != 0)

{

rem = n%10;

rev = rev \* 10 + rem ;

n = n/10 ;

}

if(rev == temp)

System.out.println("Palindrome number");

else

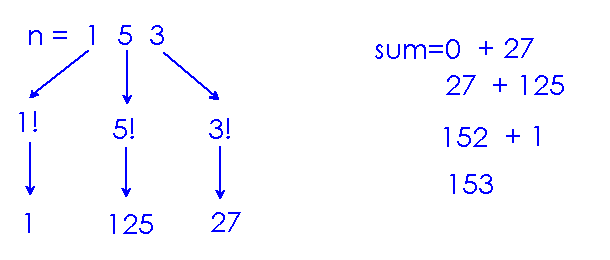
System.out.println("Not palindrome number");

}

}

Strong Number : Sum of factorials of individual digits is equals to the same number.

N = 153 🡪 1! + 5! + 3! = 1 + 125 + 27 = 153



import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("enter n value");

int n= scan.nextInt();

int temp=n , sum=0;

while(n !=0)

{

int r = n%10;

int fact=1;

for(int i =1;i<=r;i++)

{

fact=fact\*i;

}

sum=sum+fact;

n=n/10;

}

if (sum == temp)

System.out.println("the number is strong number");

else

System.out.println("the number is not strong number");

}

}

break:

* It is a keyword
* It is used to break loop execution
* It must be used inside the loop or switch.

A block terminates after execution, no need to break explicitly.

class Test

{

public static void main(String[] args)

{

if(true)

{

System.out.println("It is block");

break;

}

}

}

We can break infinite loop also.

class Test

{

public static void main(String[] args)

{

while(true)

{

System.out.println("It is loop");

break;

}

}

}

We can break the loop on condition:

class Test

{

public static void main(String[] args)

{

for (int i=1 ; i<=10 ; i++)

{

if(i==5)

{

break;

}

System.out.println("i value : " + i);

}

}

}

**Continue:**

* It is a keyword
* It is used to skip the iteration in loop execution.

class Test

{

public static void main(String[] args)

{

for (int i=1 ; i<=10 ; i++)

{

if(i==5)

{

continue;

}

System.out.println("i value : " + i);

}

}

}

class Test

{

public static void main(String[] args)

{

for (int i=1 ; i<=10 ; i++)

{

if(i==3 || i==6)

{

continue;

}

System.out.println("i value : " + i);

}

}

}

switch:

* It is conditional control statement
* It is allowed to define multiple blocks called case statements.
* Switch randomly execute a particular case depends on condition.
* After execution of case statements, we must break the switch.

class Test

{

public static void main(String[] args)

{

char ch='g';

switch(ch)

{

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

break;

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

break;

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

break;

default : System.out.println("Invalid color");

}

}

}

import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

while(true)

{

System.out.println("Arithmetic operations");

System.out.println("1. Add");

System.out.println("2. Subtract");

System.out.println("3. Multiply");

System.out.println("4. Quit");

System.out.println("Enter your choice : ");

int ch = scan.nextInt();

int a, b;

switch(ch)

{

case 1 : System.out.println("Enter 2 numbers :");

a=scan.nextInt();

b=scan.nextInt();

System.out.println("Sum : " + (a+b));

break;

case 2 : System.out.println("Enter 2 numbers :");

a=scan.nextInt();

b=scan.nextInt();

System.out.println("Difference : " + (a-b));

break;

case 3 : System.out.println("Enter 2 numbers :");

a=scan.nextInt();

b=scan.nextInt();

System.out.println("Product : " + (a\*b));

break;

case 4 : System.exit(0);

default: System.out.println("Invalid input");

}

}

}

}

Access Modifiers

* Access modifiers are used to set access permissions on members(variables and methods)
* Java supports
  + Private
  + Public
  + <package>
  + Protected

We can apply access modifiers only to the members which are having identities(names). Hence blocks don’t have access modifiers.

public class Test

{

public static int a ;

private Test()

{

// logic

}

protected void fun()

{

// logic

}

public static

{

// logic

}

}

Private:

* It is a keyword.
* ‘private’ members we can access directly with in the class.

class First

{

public static int x=10;

private static int y=20;

public static void main(String[] args)

{

System.out.println("Accessing from same(First) class");

System.out.println("x value : " + First.x);

System.out.println("y value : " + First.y);

}

}

Compiler raises error message when we try to access private members of another class directly.

Private means secured.

Hence not allowed to access one class(object) information directly from another class(object).

class First

{

public static int x=10;

private static int y=20;

}

class Second

{

public static void main(String[] args)

{

System.out.println("Accessing from another(Second) class");

System.out.println("x value : " + First.x);

System.out.println("y value : " + First.y); // Error : 'y' has private in First class

}

}

Accessing private non static members from same class – is allowed:

class First

{

public int a ;

private int b ;

First(int a , int b)

{

this.a = a ;

this.b = b ;

}

public static void main(String[] args)

{

First obj = new First(10, 20);

System.out.println("From same class : ");

System.out.println("a value is : " + obj.a);

System.out.println("b value is : " + obj.b);

}

}

Accessing the private non static variables from another class results error:

class First

{

public int a ;

private int b ;

First(int a , int b)

{

this.a = a ;

this.b = b ;

}

}

class Second

{

public static void main(String[] args)

{

First obj = new First(10, 20);

System.out.println("From another class : ");

System.out.println("a value is : " + obj.a);

System.out.println("b value is : " + obj.b); // Error : 'b' is private in class First

}

}

If the constructor is private, we cannot create object from another class:

class First

{

public First()

{

System.out.println("First class object created...");

}

}

class Second

{

private Second()

{

System.out.println("Second class object created...");

}

}

class Third

{

public static void main(String args[ ])

{

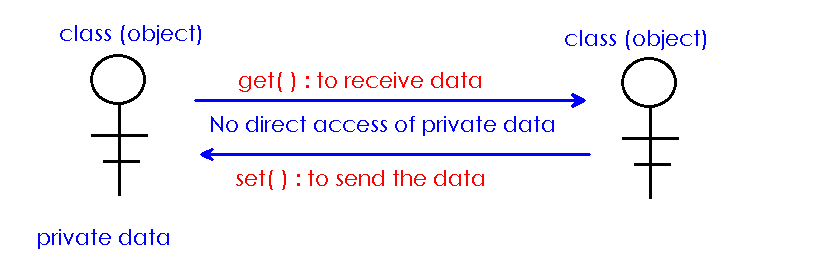
First obj1 = new First();

Second obj2 = new Second(); // Error : constructor is private

}

}

* One object cannot access the private information of another object directly.
* One object can share the private information to other object in communication.
* Communication is possible using methods.
* Java is providing 2 standard methods to send and receive the data.



* Get() method is used to receive the information of object
* Set() method is used to send the information to object.

class Employee

{

private static int id = 101;

private static String name = "amar";

}

class Access

{

public static void main(String[] args)

{

System.out.println("Employee details are : ");

System.out.println("Emp ID is : " + Employee.id); // Error :

System.out.println("Emp name is : " + Employee.name); // Error :

}

}

Access using get() methods:

class Employee

{

private static int id = 101;

private static String name = "amar";

public static int getId()

{

return Employee.id;

}

public static String getName()

{

return Employee.name;

}

}

class Access

{

public static void main(String[] args)

{

System.out.println("Employee details are : ");

System.out.println("Emp ID is : " + Employee.getId());

System.out.println("Emp name is : " + Employee.getName());

}

}

Complete code:

class First

{

private static int a = 10;

private static boolean b = true;

private static double c = 23.45;

}

class Second

{

public static void main(String[] args)

{

// Access a, b, c and print values

}

}

Set() method is used to modify the values of private variables:

class Sample

{

private static int a = 10;

public static int getA()

{

return Sample.a;

}

public static void setA(int a)

{

Sample.a = a ;

}

}

class Test

{

public static void main(String[] args)

{

System.out.println("a value : " + Sample.getA());

Sample.setA(20);

System.out.println("modified a value : " + Sample.getA());

}

}

Set() and Get() methods of Employee class:

class Employee

{

private static int id=1;

private static String name="subha";

private static double salary=2000;

public static int getId()

{

return Employee.id ;

}

public static String getName()

{

return Employee.name ;

}

public static double getSalary()

{

return Employee.salary ;

}

public static void setId(int a)

{

Employee.id=a;

}

public static void setName(String b)

{

Employee.name=b;

}

public static void setSalary(double c)

{

Employee.salary=c;

}

}

class Test

{

public static void main(String[]args)

{

System.out.println("id is:"+Employee.getId());

System.out.println("name is:"+Employee.getName());

System.out.println("salary is:"+Employee.getSalary());

Employee.setId(2);

Employee.setName("sai");

Employee.setSalary(3000);

System.out.println("modified id is:"+Employee.getId());

System.out.println("modified name is:"+Employee.getName());

System.out.println("modified salary is:"+Employee.getSalary());

}

}

Set() and get() methods for non static variables:

class Sample

{

private int a ;

Sample(int a)

{

this.a = a ;

}

public int getA()

{

return this.a ;

}

public void setA(int a)

{

this.a = a;

}

}

class Test

{

public static void main(String[] args)

{

Sample obj = new Sample(10);

System.out.println("a value is : " + obj.getA());

obj.setA(20);

System.out.println("a value is : " + obj.getA());

}

}

Account operations:

class Account

{

private int balance ;

Account(int balance)

{

this.balance = balance;

}

public int getBalance()

{

return this.balance ;

}

void withdraw(int amount)

{

if(amount <= this.balance)

{

System.out.println("collect cash : " + amount);

this.balance = this.balance - amount;

}

else

System.out.println("Low balance error");

}

}

class Bank

{

public static void main(String[] args)

{

Account acc = new Account(5000);

System.out.println("Balance is : " + acc.getBalance());

acc.withdraw(3000);

System.out.println("Final Balance is : " + acc.getBalance());

}

}

class Test

{

static Test obj = new Test();

static

{

Test.obj.fun();

}

public static void main(String[] args)

{

Test.obj.fun();

}

void fun()

{

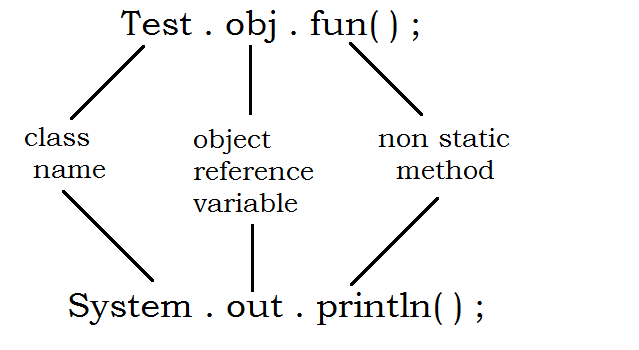
System.out.println("fun....");

}

}

System.out.println():

* Println() is a non static method.
* To access println() method, object is required.
* “out” is a pre-defined object reference variable.
* “out” is static variable in System class.
  + System.out.println();



class Laptop

{

void data()

{

System.out.println("Copying data....");

}

}

class Faculty

{

static Laptop password = new Laptop();

}

class Student

{

public static void main(String[] args)

{

Faculty.password.data();

}

}

class Bike

{

void ride()

{

System.out.println("Riding bike....");

}

}

class Friend

{

static Bike key = new Bike();

}

class You

{

public static void main(String args[])

{

Friend.key.ride();

}

}

class Account

{

void withdraw()

{

System.out.println("Withdrawing amount from account...");

}

}

class Father

{

static Account atm = new Account();

}

class You

{

public static void main(String args[ ])

{

Father.atm.withdraw();

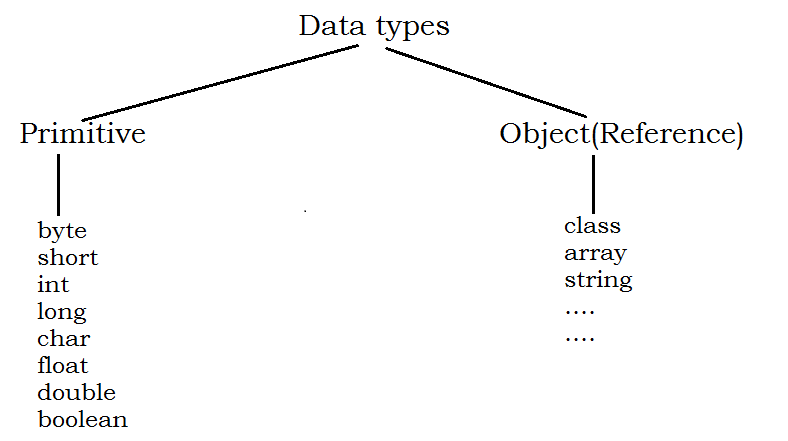
}

}

Data types

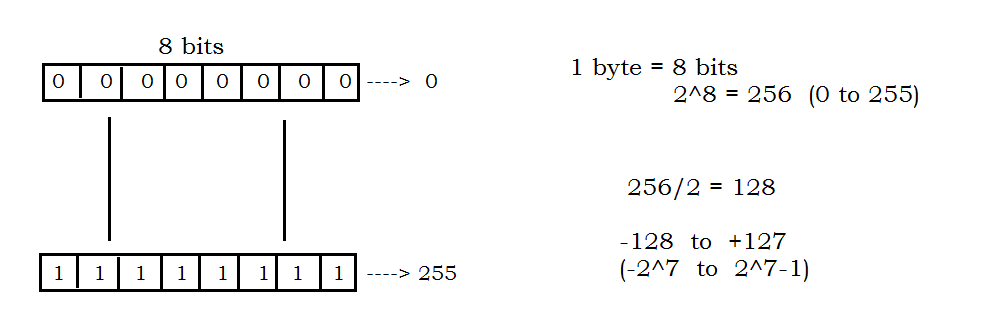
* In the declaration of variable, we need to specify its data type.
  + data\_type name = value ;
* Data type describes
  + What is the size of memory location?
  + What type of data is allowed to store?
* For example
  + int a ;
  + The size is ‘4’ bytes
  + Only integer type data is allowed.

Classification:

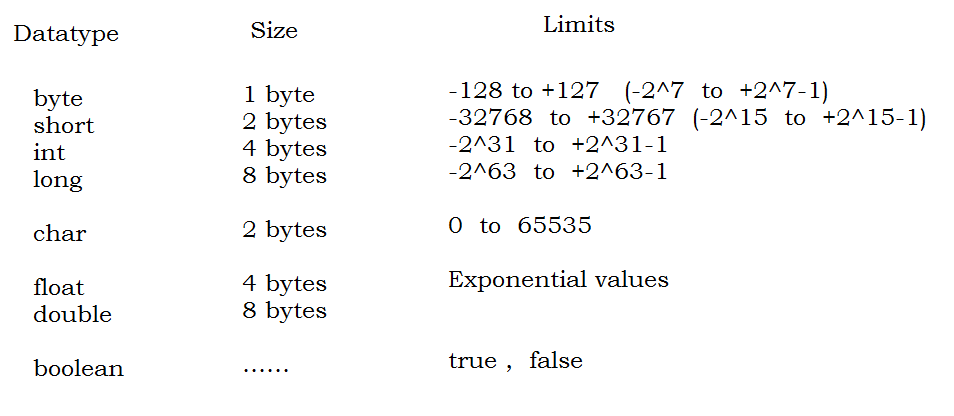


Primitive types:

* We represent the memory in bytes
* 1 byte = 8 bits
* We store the data always in binary format.

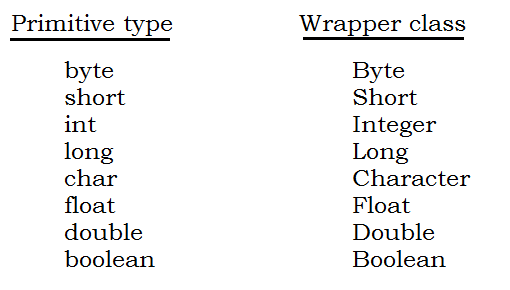


**Limits of data types:**

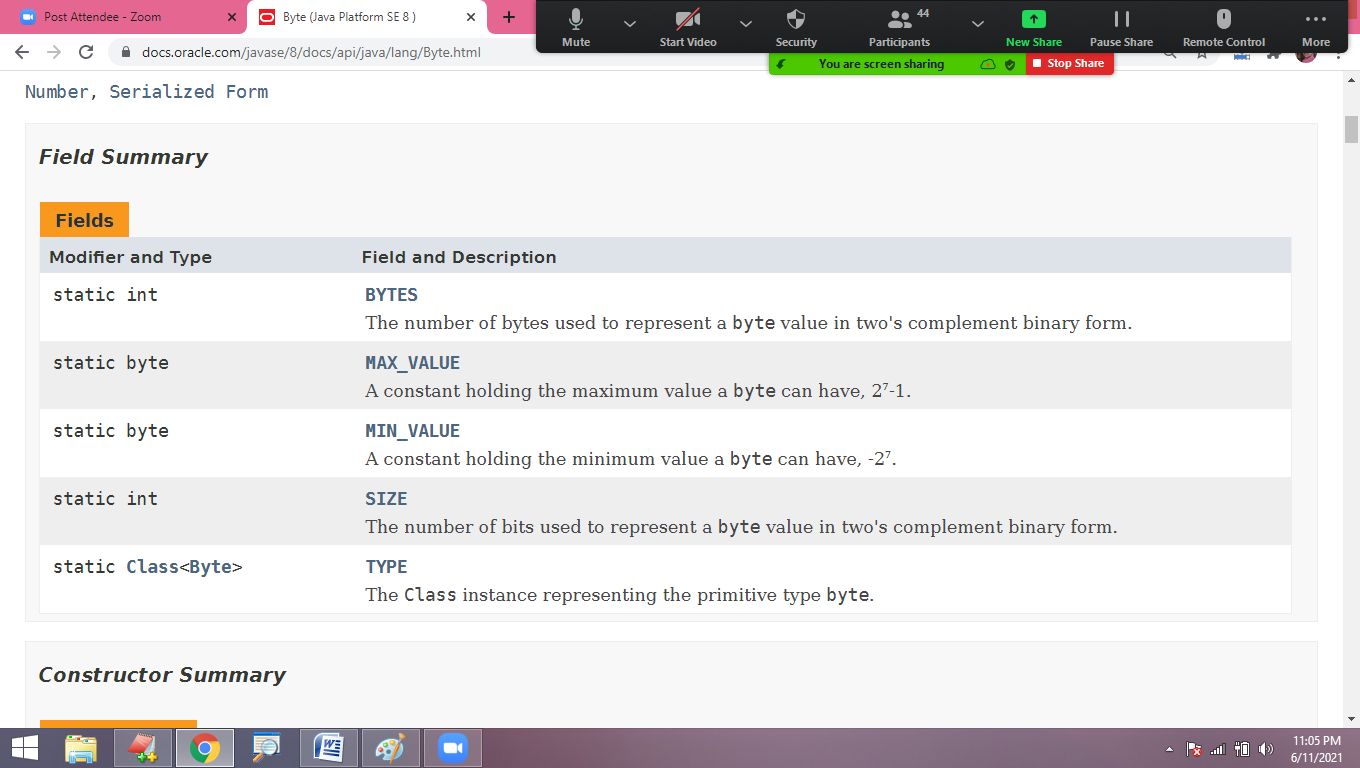


**Wrapper classes:**

* For every primitive type, there is a corresponding wrapper class.
* Wrapper classes providing pre-defined variables of static type.
* These variables represents
  + Size of data type
  + Limits of data type



The Byte class holding static variables as shown below:



/\*

class Byte

{

public static byte MIN\_VALUE;

public static byte MAX\_VALUE;

public static int SIZE;

}

\*/

class Limits

{

public static void main(String[] args)

{

System.out.println("byte min value : " + Byte.MIN\_VALUE);

System.out.println("byte max value : " + Byte.MAX\_VALUE);

System.out.println("byte size in bits : " + Byte.SIZE);

}

}

All wrapper classes having the same set of fields(variables):

class Limits

{

public static void main(String[] args)

{

System.out.println("long min value : " + Long.MIN\_VALUE);

System.out.println("long max value : " + Long.MAX\_VALUE);

System.out.println("long size in bits : " + Long.SIZE);

}

}

class Limits

{

public static void main(String[] args)

{

System.out.println("char min value : " + (int)Character.MIN\_VALUE);

System.out.println("char max value : " + (int)Character.MAX\_VALUE);

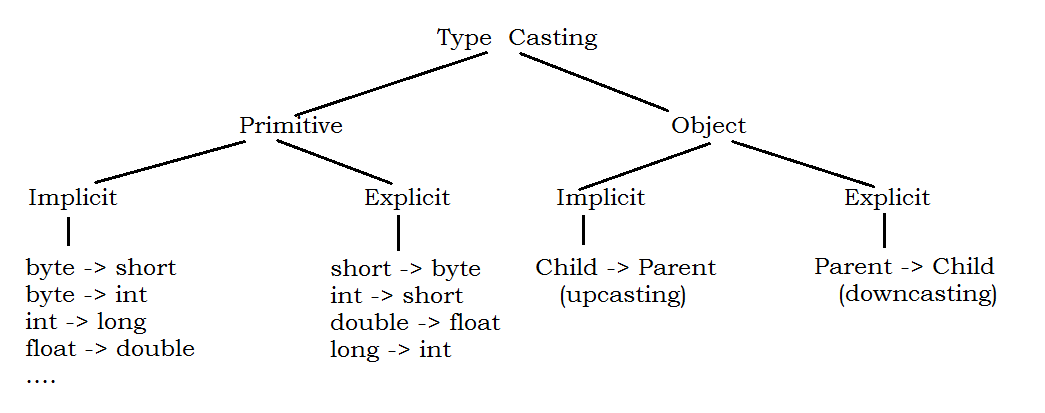
System.out.println("char size in bits : " + Character.SIZE);

}

}

**Type casting:**

* Conversion of data from one type to another type.
* Type casting in java either implicit or explicit.
* Implicit – auto conversion of data
* Explicit – manual conversion of data



class Implicit

{

public static void main(String[] args)

{

byte x = 10 ;

int y = x ; // auto conversion

System.out.println("y value is : " + y);

}

}

class Explicit

{

public static void main(String[] args)

{

int x = 10 ;

byte y = (byte)x ; // manual conversion

System.out.println("y value is : " + y);

}

}

In the conversion of character type to integer, we will get ASCII values or Symbols:

class Implicit

{

public static void main(String[] args)

{

char x = 'A' ;

int y = x ; // auto

System.out.println("y value is : " + y);

}

}

class Explicit

{

public static void main(String[] args)

{

int x = 36 ;

char y = (char)x ; // manual conversion

System.out.println("y value is : " + y);

}

}

Character data type:

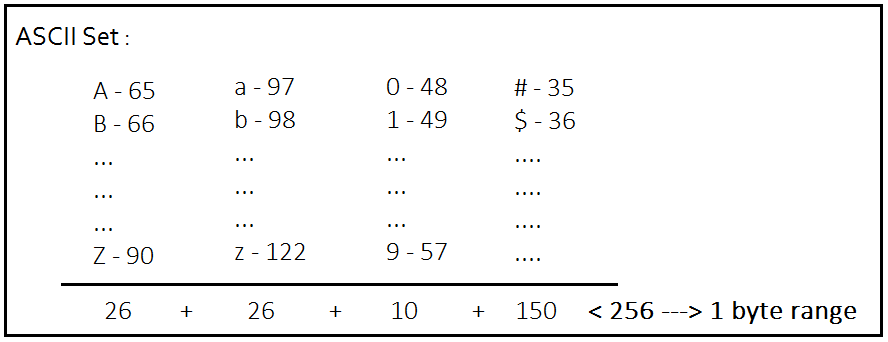
* Character type can store character, symbols and digits.
* Character can be represented using ‘single quotes’.
* Character type can also holds integers.
* Character limits representing with integers (ASCII values).
* Character size is 2 bytes in Java and the range of 0 to 65535.

**Why char data type limits discussion in integers?**

* Character is an integer in the memory.
* Character can be represented with constant integer in Character system.
* Character System is the representation of all the symbols of one language with constant integer values.
* One of the popular character systems is ‘ASCII’

**How can we store a symbol into 1 byte memory in C or C++?**

* A language at most having 256 symbols(1 byte range)
* According to ASCII character system, any symbol in any language is assigned with value within range of 0 to 255.
* Hence we can store the symbol corresponding value in 1 byte.



**Why character occupy 2 bytes memory in java and .net where as it occupies only 1 byte in C and C++?**

* Java & .net apps are web apps.
* Web app need to represent more than one language character set a time. Hence it occupies 2 bytes.
* C & C++ are platform dependent. These can represent only one language at a time.

class IntToByte

{

public static void main(String[] args)

{

int i = 130 ;

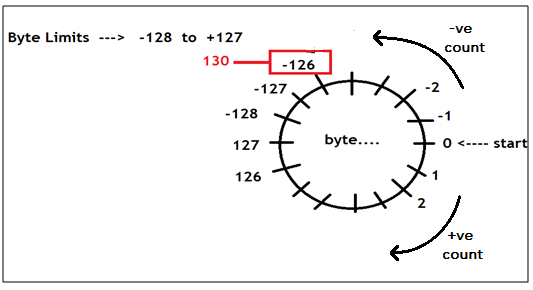
byte b ;

b = (byte)i ;

System.out.println("b value : "+b);

}

}



class ExplicitCast

{

public static void main(String[] args)

{

int i1 = 256 , i2 = 512 , i3 =1024 ;

byte b1, b2, b3 ;

b1 = (byte)i1 ;

b2 = (byte)i2 ;

b3 = (byte)i3 ;

System.out.println("b1 value : "+b1);

System.out.println("b2 value : "+b2);

System.out.println("b3 value : "+b3);

}

}

class ExplicitCast

{

public static void main(String[] args)

{

int i1 = -131 ;

byte b1 ;

b1 = (byte)i1 ;

System.out.println("b1 value : "+b1);

}

}

**Using type casting, we can conserve/save the memory in the application….**

class DataTypes

{

public static void main(String[] args)

{

byte a=10 , b=20 ;

// int c = a+b ; /\* Allowed but occupies 4 byte memory \*/

byte c = (byte)(a+b) ;

System.out.println("c value : "+c);

}

}

|  |  |  |
| --- | --- | --- |
| **float type** | **Size in bytes** | **Limits** |
| float | 4 | ±3.40282347E+38F |
| double | 8 | ±1.79769313486231570E+308 |

/\*

int / int --> int

int / float --> float

float / int--> float

float / float --> float

\*/

class DataTypes

{

public static void main(String[] args)

{

int x=5 , y=2 ;

int z = x/y ;

System.out.println("z value : "+z);

}

}

class DataTypes

{

public static void main(String[] args)

{

int x=5 , y=2 ;

float z = x/y ;

System.out.println("z value : "+z);

}

}

class DataTypes

{

public static void main(String[] args)

{

int x=5 , y=2 ;

float z = (float)x/y ;

System.out.println("z value : "+z);

}

}

Boolean data type:

* A data type which stores true or false.
* It doesn’t have size.
* Boolean values are pre-defined keywords.

class BooleanType

{

public static void main(String args[ ])

{

System.out.println("int size : "+Integer.SIZE);

System.out.println("boolean size : "+Boolean.SIZE); // Error :

}

}

class BooleanType

{

static boolean b ;

public static void main(String args[ ])

{

System.out.println("boolean default value : "+BooleanType.b);

}

}

class Test

{

public static void main(String args[ ])

{

System.out.println("Is 6 perfect number : "+Test.isPerfect(6));

}

static boolean isPerfect(int n)

{

int sum=0 ;

for(int i=1 ; i<n ; i++)

{

if(n%i==0)

sum=sum+i;

}

if(sum==n)

return true;

else

return false;

}

}

**Wrapper classes**

* Wrapper classes are pre-defined.
* Wrapper classes are used to perform data conversions with pre-defined methods.
* Conversions are:
  + Primitive -> Object (Boxing)
  + Object -> Primitive (Un-boxing)
  + Primitive -> String
  + String -> Primitive
  + Object -> String
  + String -> Object

For every primitive type, there is a corresponding wrapper class.

1. byte -> Byte
2. short -> Short
3. int -> Integer
4. long -> Long
5. char -> Character
6. boolean -> Boolean
7. float -> Float
8. double -> Double

**Integer conversions:**

Boxing : Converting primitive type to Object type (int -> Integer)

/\*

class Integer

{

public static Integer valueOf(int i) ;

}

\*/

class Boxing

{

public static void main(String[] args)

{

int x = 10 ;

Integer y = Integer.valueOf(x);

System.out.println("y value : " + y);

}

}

Primitive To String :

* Take primitive integer value
* Convert into String using Integer class method.

/\*

class Integer

{

public static String toString(int i)

{

Returns a String object representing the specified integer.

}

}

\*/

class PrimitiveToString

{

public static void main(String[] args)

{

int x = 10 ;

String y = Integer.toString(x);

System.out.println("y value is : " + y);

}

}

Un-boxing:

* Converting Object type to Primitive type is called un-boxing.
* Integer class is providing the pre-defined method for this conversion.
* We need to input Object type for this method to convert.
* We will get primitive type after this conversion.

/\*

class Integer

{

public int intValue()

{

Returns the value of this Integer as an int.

}

}

\*/

class UnBoxing

{

public static void main(String[] args)

{

int x = 10 ;

Integer y = Integer.valueOf(x); // 'y' is Object

int z = y.intValue(); // unboxing

System.out.println("z value is : " + z);

}

}

Convert Object to String:

* Using pre-defined non static method, we can convert Object type into String type.

/\*

class Integer

{

public String toString()

{

Returns a String object representing this Integer's value.

}

}

\*/

class ObjectToString

{

public static void main(String[] args)

{

int x=10;

Integer y = Integer.valueOf(x);

String z = y.toString();

}

}

Complete the code:

class Conversions

{

public static void main(String[] args)

{

String a = "10";

// convert 'a' into int

// convert int into Integer

// convert Integer into int

// convert int into String

}

}

/\* class Integer

{

public static int parseInt(String s) throws NumberFormatException

{

If the input is valid int(string form) then it converts

If the input is not valid, raises Exception(Runtime error)

}

} \*/

class StringToPrimitive

{

public static void main(String[] args)

{

String s1 = "10";

int x = Integer.parseInt(s1);

System.out.println("x value is : " + x);

String s2 = "abc";

int y = Integer.parseInt(s2); // Exception : Invalid input given

System.out.println("y value is : " + y);

}

}

/\*

class Integer

{

public static Integer valueOf(String s) throws NumberFormatException

}

\*/

class StringToObject

{

public static void main(String[] args)

{

String s = "10";

Integer i = Integer.valueOf(s);

}

}

class Conversions

{

public static void main(String[] args)

{

int a = 10 ;

String b =

Integer c =

int d =

Integer e =

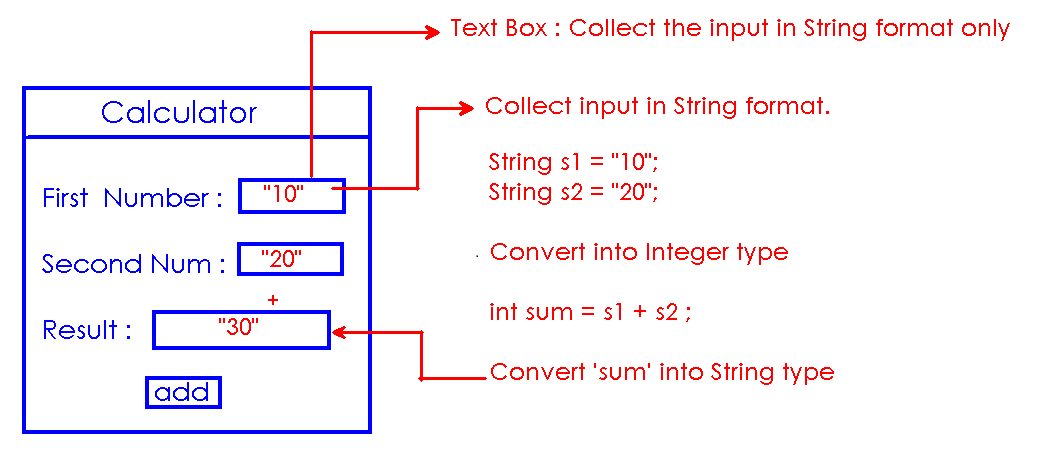
String f =

int g =

}

}

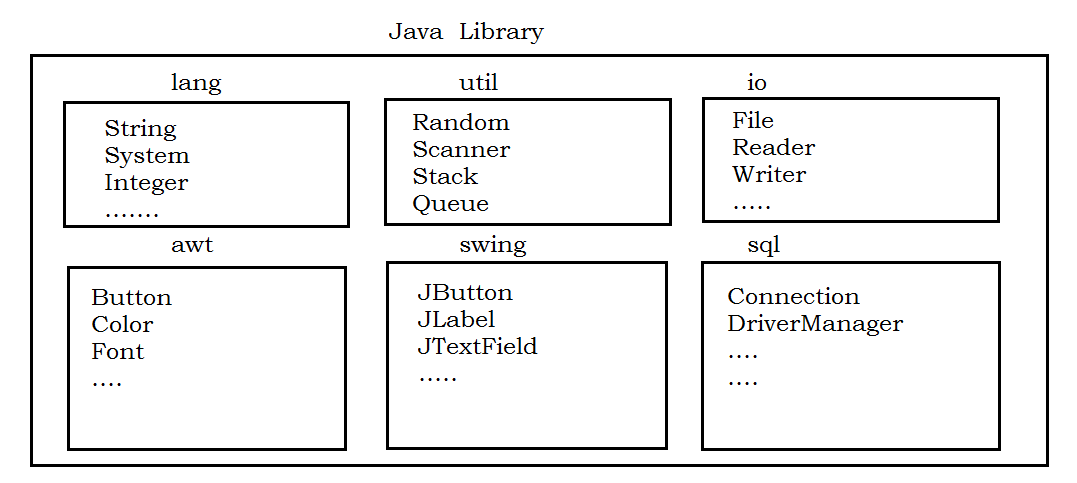
Where we use String type conversions?



**Packages**

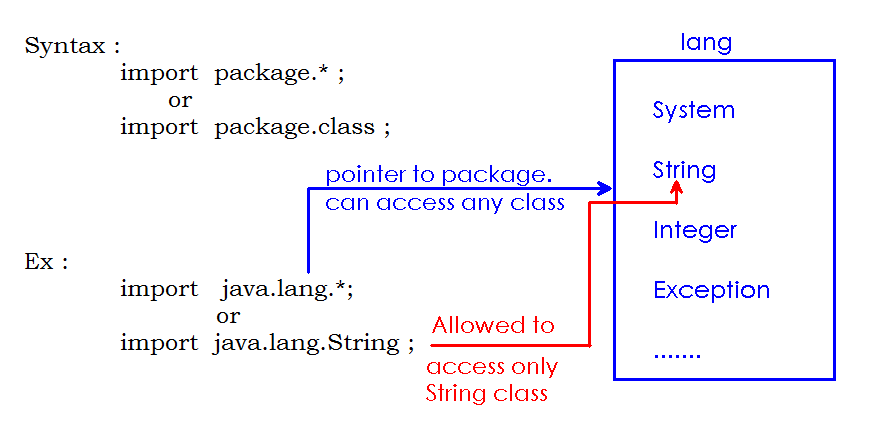
Package:

* ‘package’ is a java folder or directory.
* ‘package’ is a collection of related classes.
* Java library is a collection of ‘packages’
* Naming convention of package is ‘single word in lower case’
  + lang, util, awt, io, swing, sql, servlet……



import:

* It is a keyword.
* It is used to connect the packages in java application.
* We can access classes in package by importing that package.



**Working with pre-defined packages:**

**lang:**

* It is called “default” package.
* “lang” package classes directly available into every java application.
* No need to import “lang” package to work with classes.

class StringToObject

{

public static void main(String[] args)

{

String s = "10";

Integer i = Integer.valueOf(s);

System.out.println("i value : " + i);

}

}

**Note:** In the above application, all classes(String, Integer and System) belongs to lang package only.

util package:

* Utility classes and Collection frame work classes belongs to util package only.

**Random class:**

* It is belongs to util package.
* We must import the class to use.
* Random class is used to generate random numbers in java application.

class Demo

{

public static void main(String[] args)

{

Random rand = new Random(); // Error : cannot find symbol

}

}

We need to import Random class as follows:

import java.util.Random;

class Demo

{

public static void main(String[] args)

{

Random rand = new Random();

}

}

Using non static method of random class, we can generate Random numbers:

/\*

class Random

{

public int nextInt()

{

returns a random integer value

}

}

\*/

import java.util.Random;

class Demo

{

public static void main(String[] args)

{

Random rand = new Random();

int x = rand.nextInt();

System.out.println("x value is : " + x);

int y = rand.nextInt();

System.out.println("y value is : " + y);

int z = rand.nextInt();

System.out.println("z value is : " + z);

}

}

/\*

class Random

{

public int nextInt()

{

returns a random integer value

}

}

\*/

import java.util.Random;

class Demo

{

public static void main(String[] args)

{

Random rand = new Random();

for (int i=1 ; i<=10 ; i++)

{

int n = rand.nextInt();

System.out.println(n);

}

}

}

/\*

class Random

{

public int nextInt(int limit)

{

Generate a random interger from 0 to limit

}

}

\*/

import java.util.\*;

class Demo

{

public static void main(String[] args)

{

Random rand = new Random();

for (int i=1 ; i<=10 ; i++)

{

int n = rand.nextInt(100);

System.out.println(n);

}

}

}

Display 10 random numbers from 30 to 70:

import java.util.Random;

class Demo

{

public static void main(String[] args)

{

Random rand = new Random();

int count=0 ;

while(count <=10)

{

int n = rand.nextInt(70);

if(n>30)

{

System.out.println(n);

++count;

}

}

}

}

java.util.Scanner:

* It is used to read the input from the user while application is running.
* While creating Scanner class object, we must specify the input source object.
* java.lang.System class is providing pre-defined object of Keyboard(input stream)

class System

{

public static PrintStream out ;

It is Monitor object

public static InputStream in;

It is Keyboard object

}

* “in” is keyboard object
* “in” is static, access using class name(System)

Create Object:

import java.util.Scanner;

class Demo

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

}

}

Scanner class is providing pre-defined methods to read different types of data.

import java.util.Scanner;

class Demo

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter an integer : ");

int n = scan.nextInt();

System.out.println("n value is : " + n);

}

}

Code to write:

* Read 2 numbers using Scanner
* Add them
* Print the Sum

import java.util.Scanner;

class Demo

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter 2 numbers : ");

int a = scan.nextInt();

int b = scan.nextInt();

int c = a+b;

System.out.println("sum value is : " + c);

}

}

Code to write:

* Read lower limit into ‘m’
* Read upper limit into ‘n’
* Generate 10 Random numbers between m to n

import java.util.Scanner;

import java.util.Random;

class Demo

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

Random rand = new Random();

System.out.println("Enter Lower limit : ");

int m = scan.nextInt();

System.out.println("Enter Upper limit : ");

int n = scan.nextInt();

int count=0;

while(count <= 10)

{

int num = rand.nextInt(n);

if(num>m)

{

System.out.println(num);

++count;

}

}

}

}

Code to write:

* Read 2 numbers
* Construct object
* Display values

import java.util.Scanner;

class Demo

{

int a, b;

Constructor()

{

}

public static void main(String[] args)

{

Read a, b values

Create object by passing a, b values

Call display() method

}

display()

{

Display a, b values...

}

}

How to create Custom packages?

* ‘package’ is a keyword
* ‘package’ statement must be the first statement in java source file.
* ‘package’ keyword is used to create Custom packages.

package nit;

import java.lang.String;

import java.lang.System;

class First

{

public static void main(String[] args)

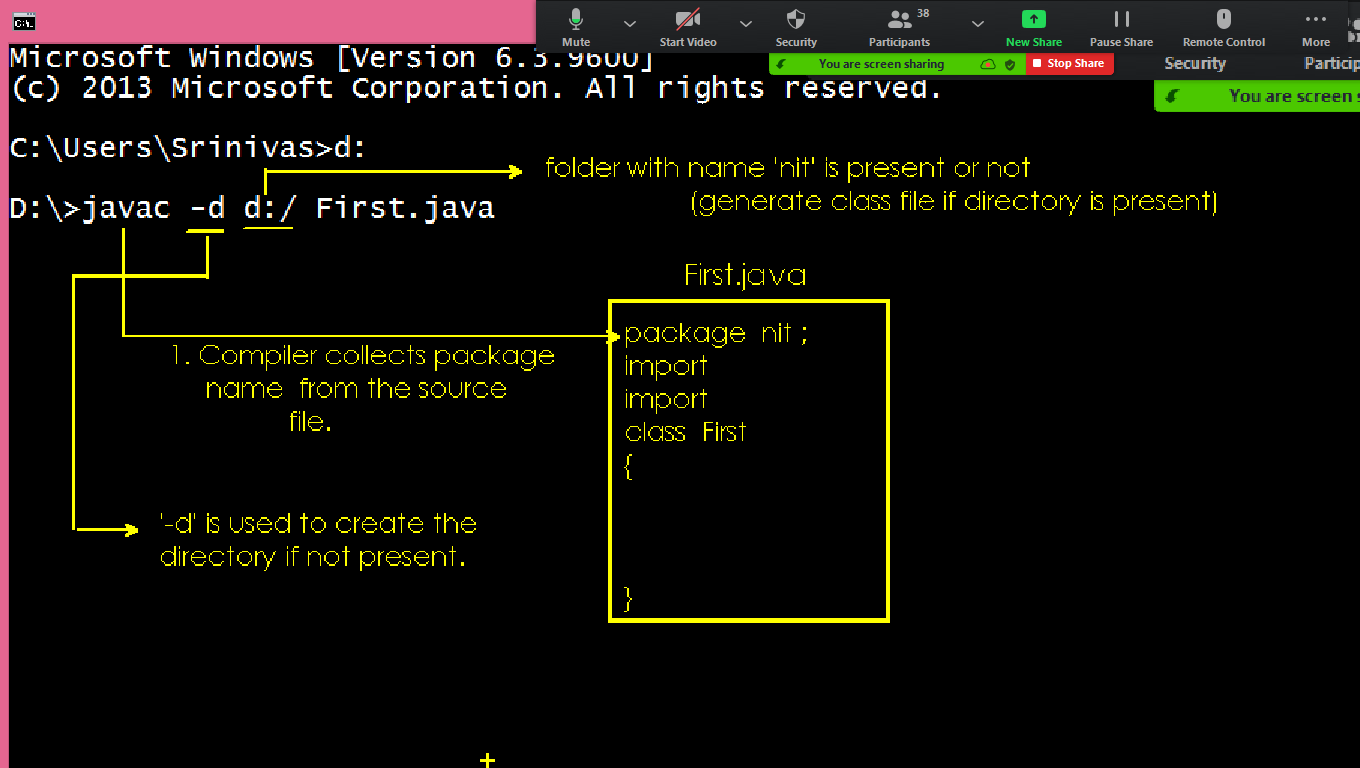
{

System.out.println("Hello World!");

}

}

Compiling the above code:



Connecting same package classes:

* We can access the classes from same package directly.
* Default access modifier in java is “package” level
* “import” statement is not required to access.

First.java:

package nit;

class First

{

static void m1()

{

System.out.println("First class m1");

}

void m2()

{

System.out.println("First class m2");

}

}

Second.java:

package nit ;

class Second

{

public static void main(String[] args)

{

System.out.println("Second class main...");

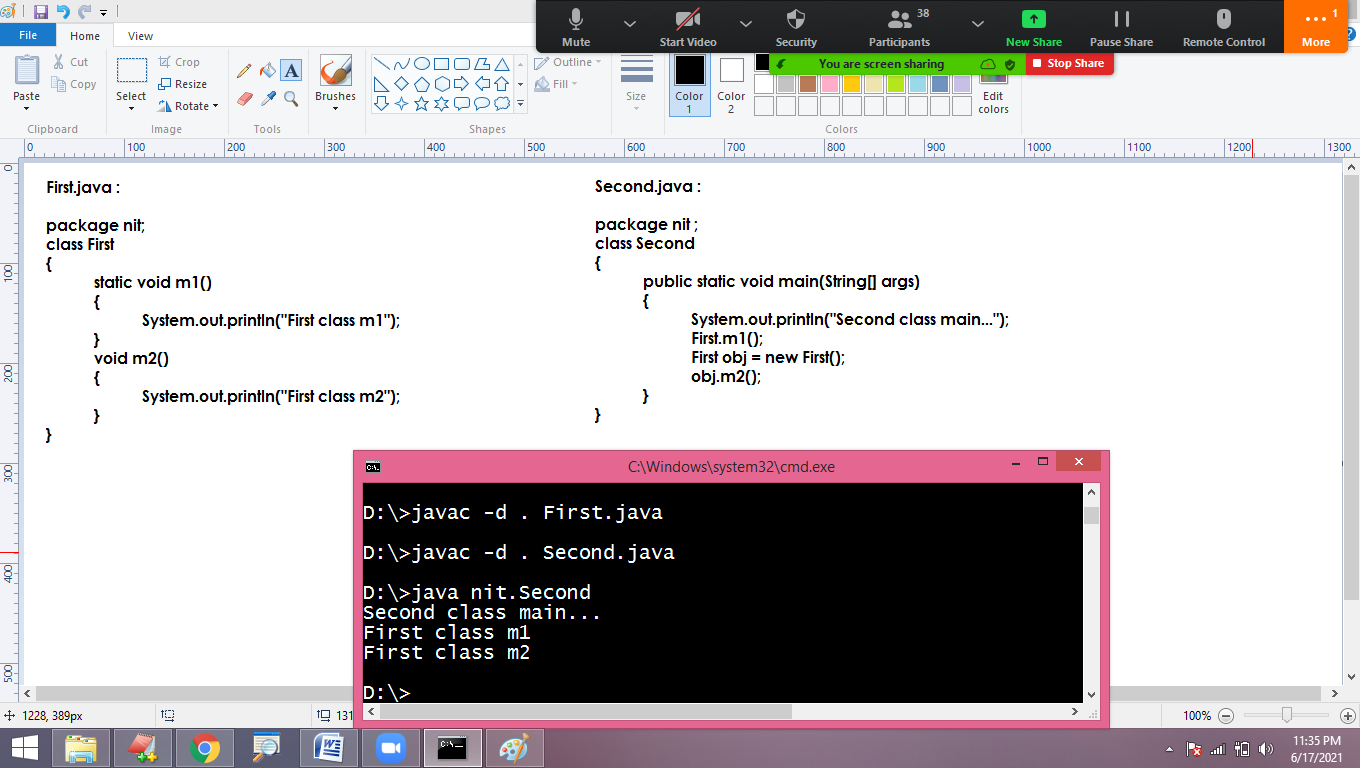
First.m1();

First obj = new First();

obj.m2();

}

}



Connecting different package classes:

* To access the members of another package, we must import class from that package.
* Only public members can access from another package.

First.java:

package p1;

public class First

{

public static void m1()

{

System.out.println("First class m1");

}

public void m2()

{

System.out.println("First class m2");

}

}

Second.java :

package p2 ;

import p1.First ;

class Second

{

public static void main(String[] args)

{

System.out.println("Second class main...");

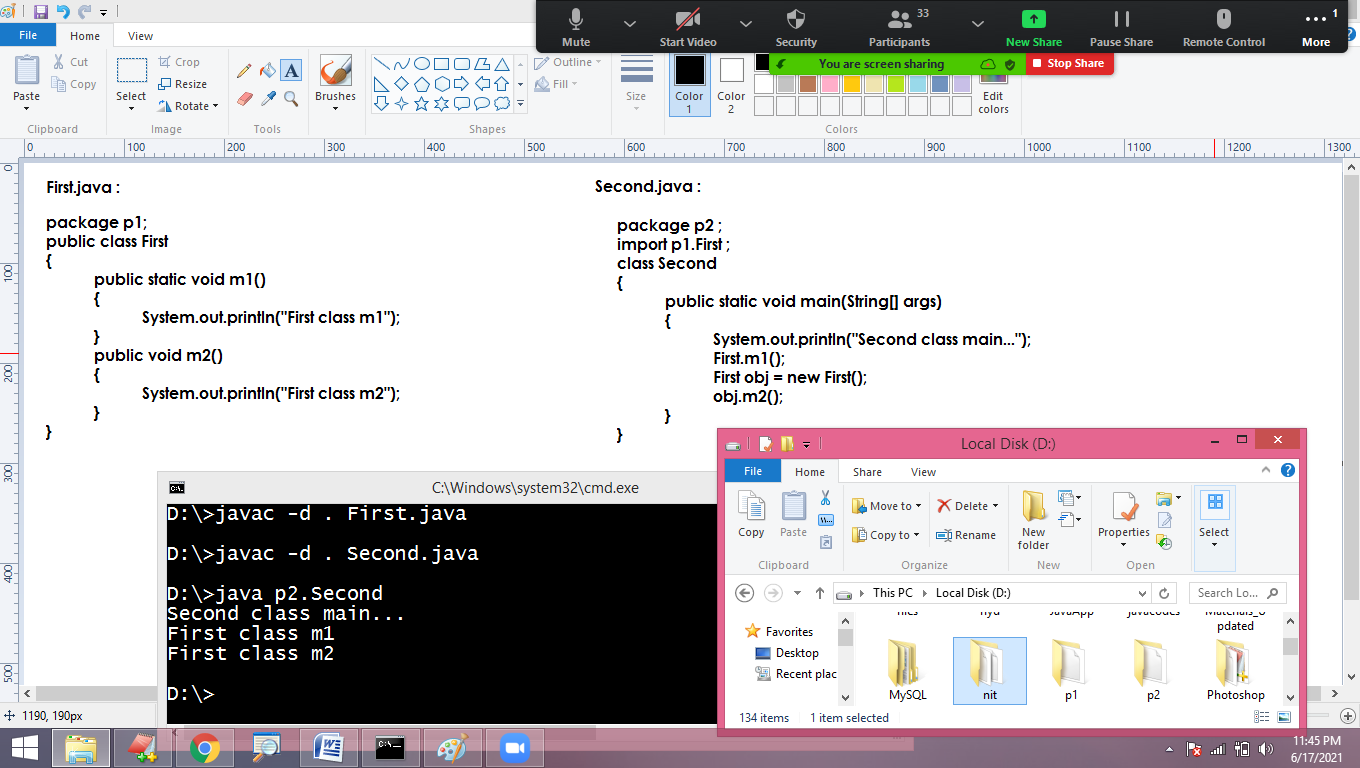
First.m1();

First obj = new First();

obj.m2();

}

}



**Fully Qualified Name:**

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

import java.util.Random;

class Demo

{

public static void main(String[] args)

{

Random rand = new Random();

}

}

Or

class Demo

{

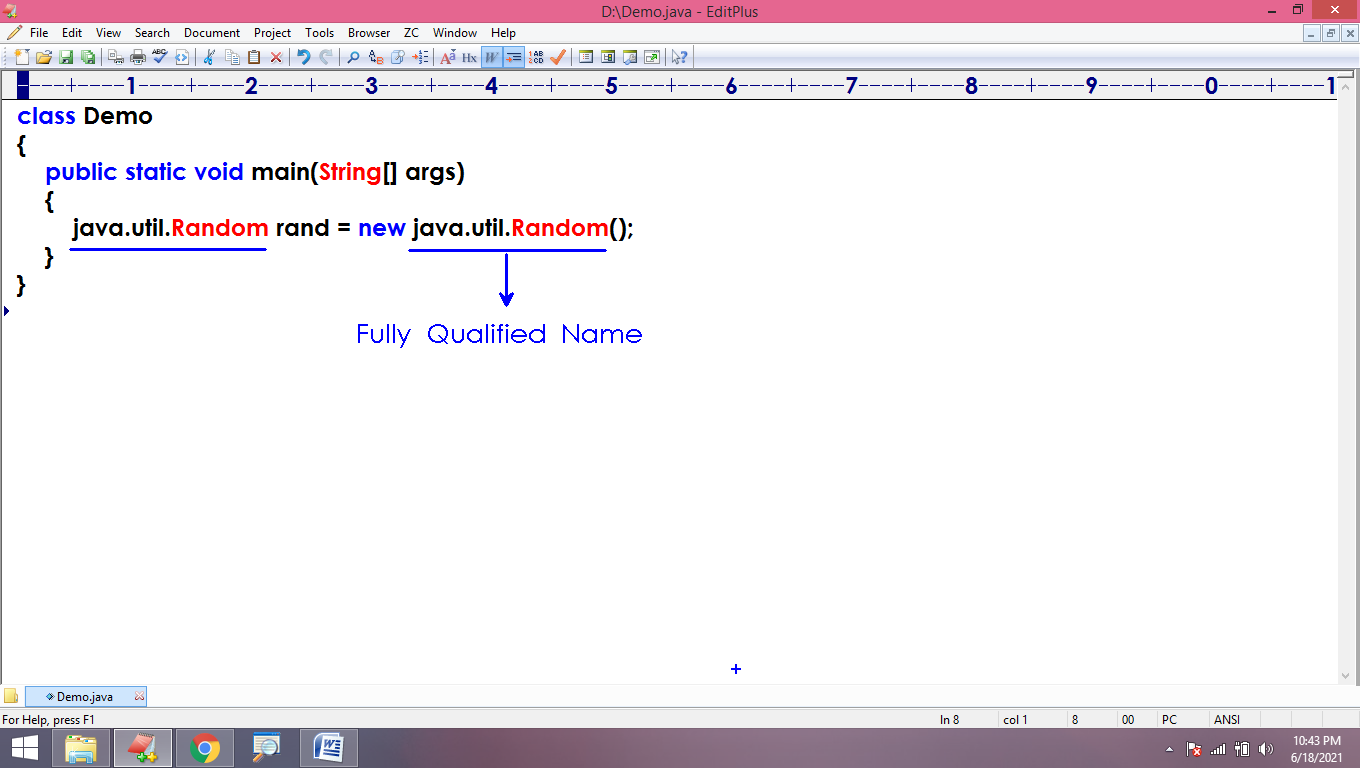
public static void main(String[] args)

{

java.util.Random rand = new java.util.Random();

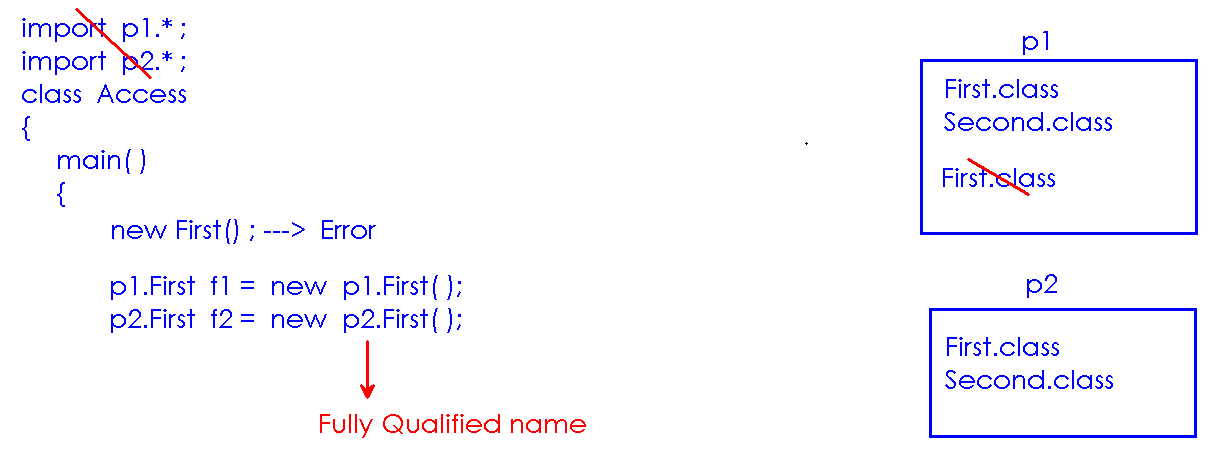
}

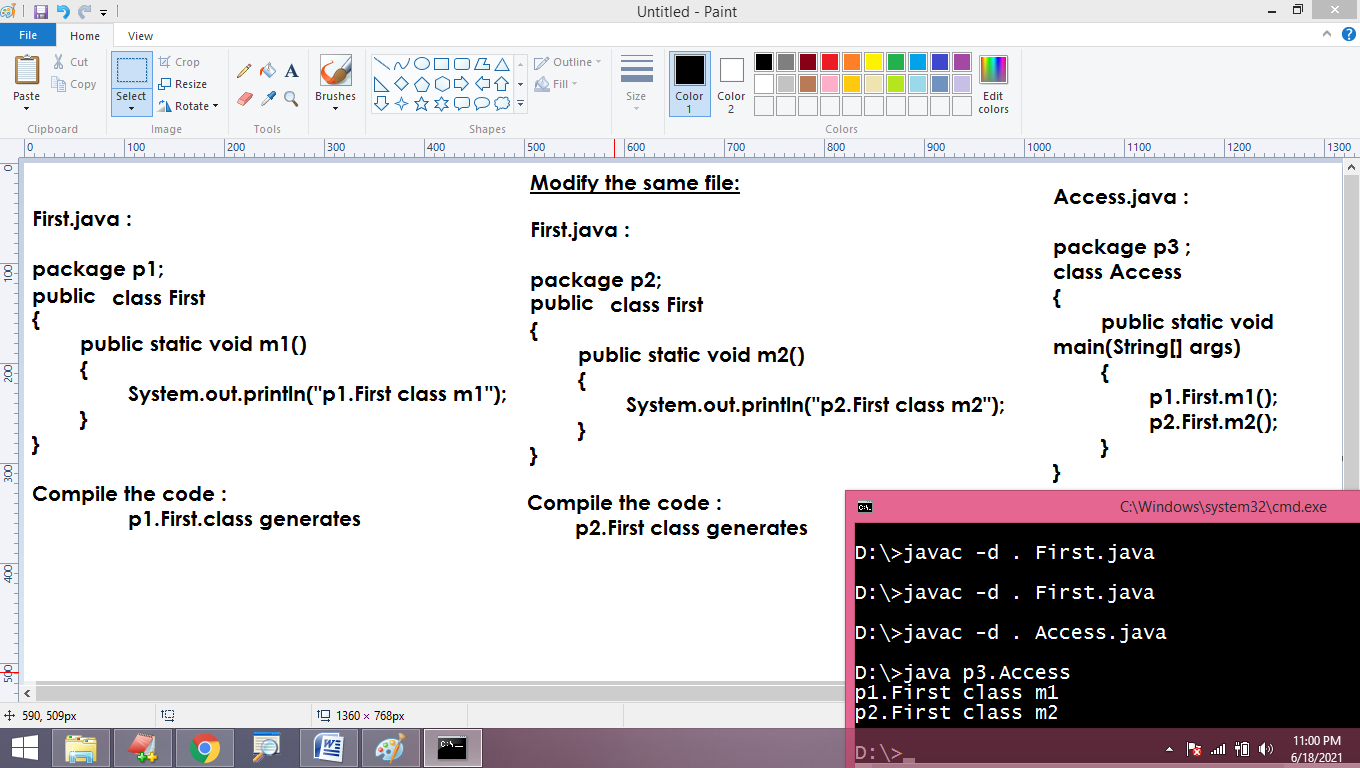
}



**When we use Fully Qualified Name?**

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





**Sub packages:**

* Defining a package inside another package.
* Compiler is responsible of creating all required packages using ‘-d’ in the process of compilation.
* If the package is present, it generates the class file automatically.

First.java:

package p1.p2;

public class First

{

public static void m1()

{

System.out.println("First class m1");

}

public void m2()

{

System.out.println("First class m2");

}

}

Second.java :

package p3.p4;

import p1.p2.First;

class Second

{

public static void main(String[] args)

{

System.out.println("Second class main");

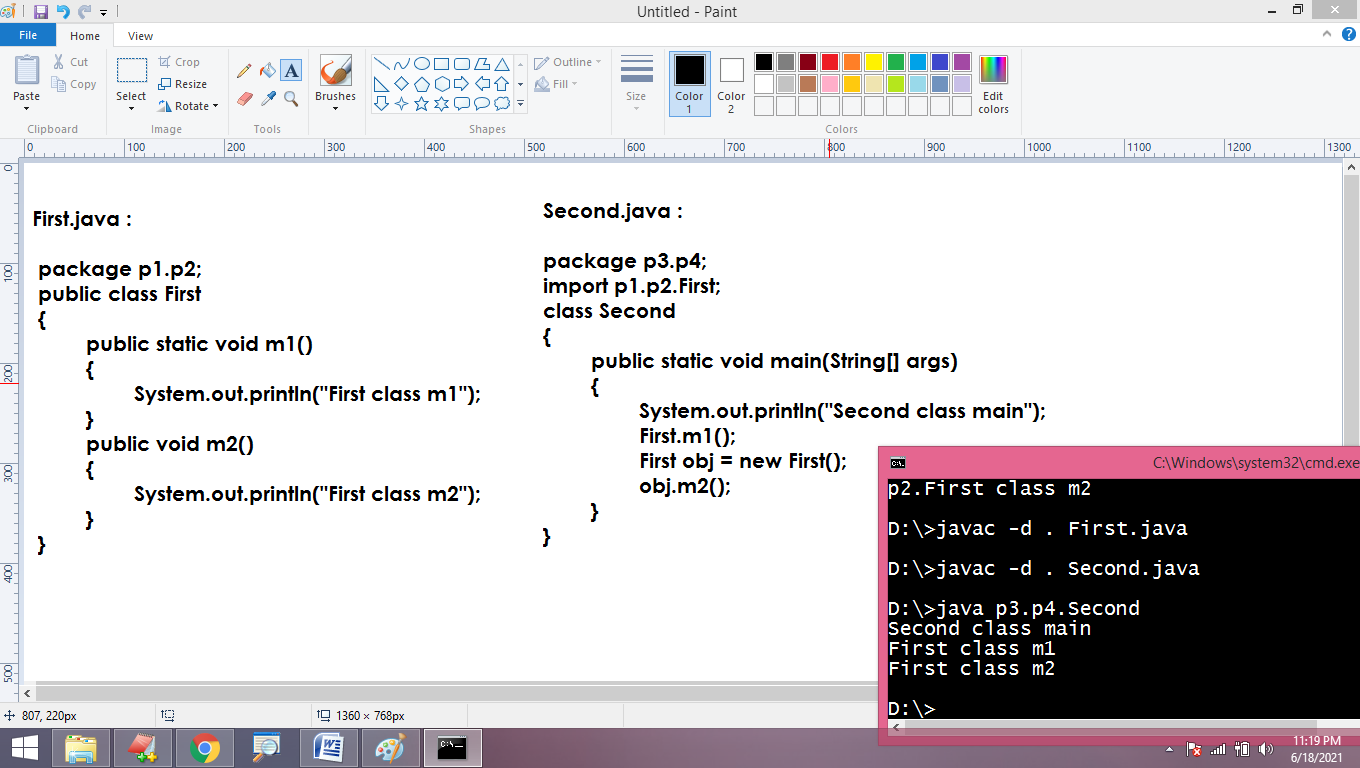
First.m1();

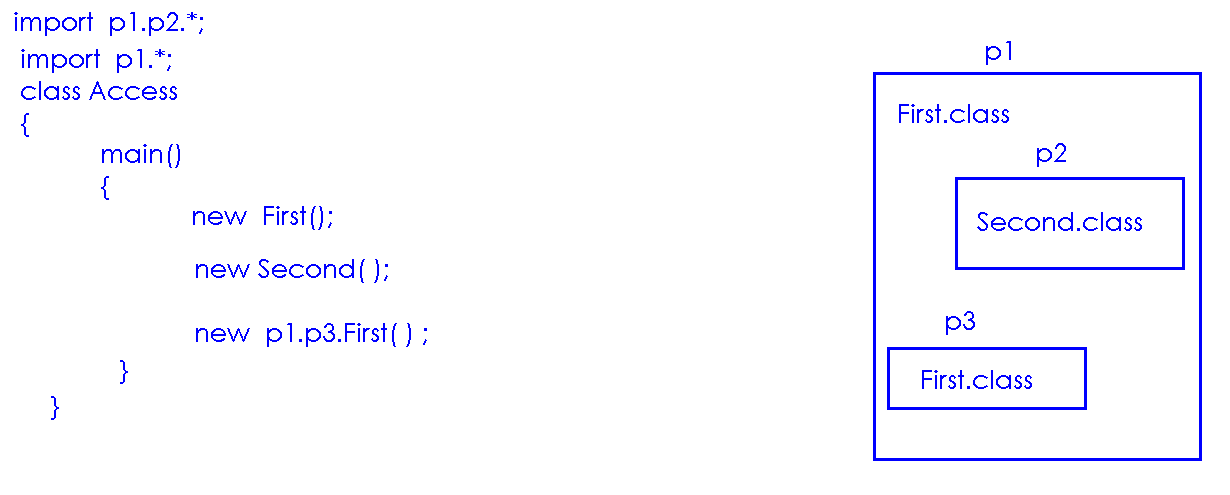
First obj = new First();

obj.m2();

}

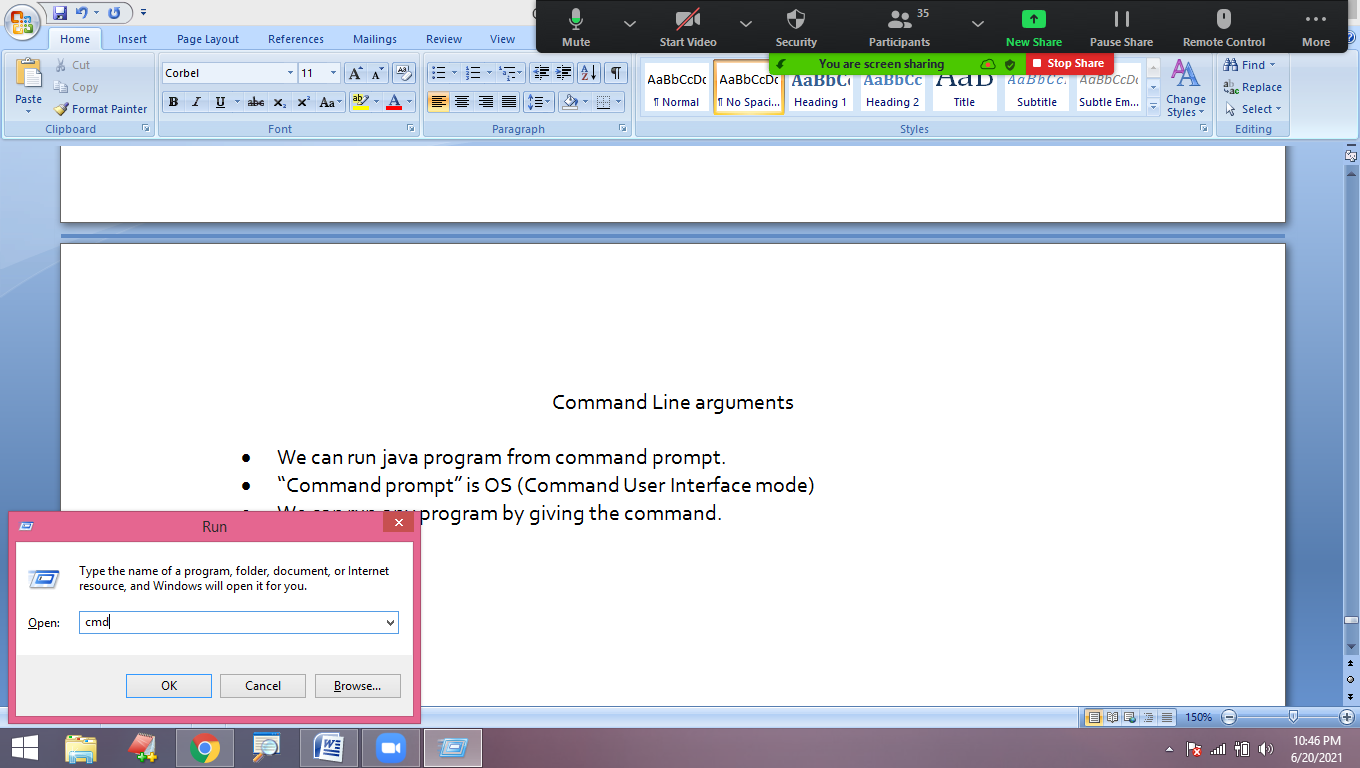
}





Command Line arguments

* We can run java program from command prompt.
* “Command prompt” is OS (Command User Interface mode)
* We can run any program by giving the command.

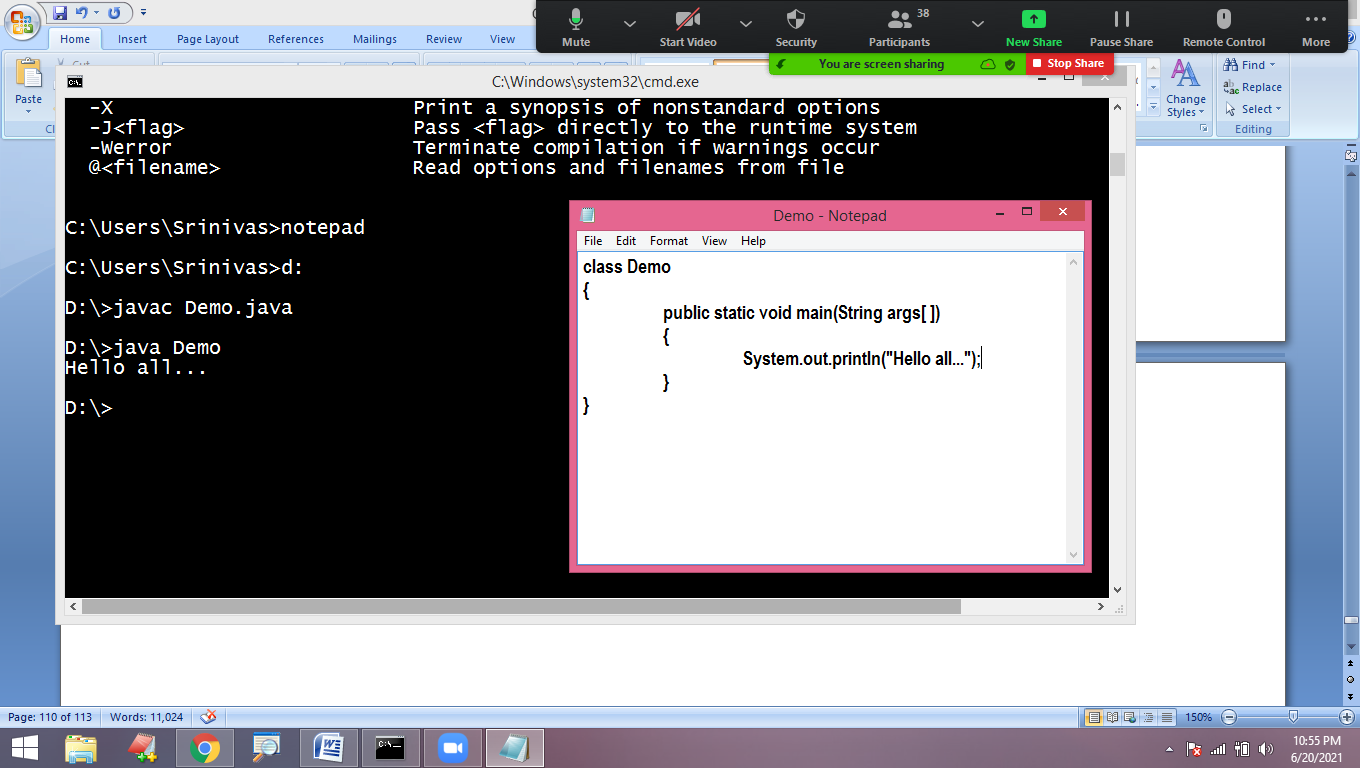




* “javac” and “java” are commands to open “Compiler” and “JVM”
* By using “javac”, we can compile the java code from command prompt
* By using “java”, we can run the java code from command prompt.

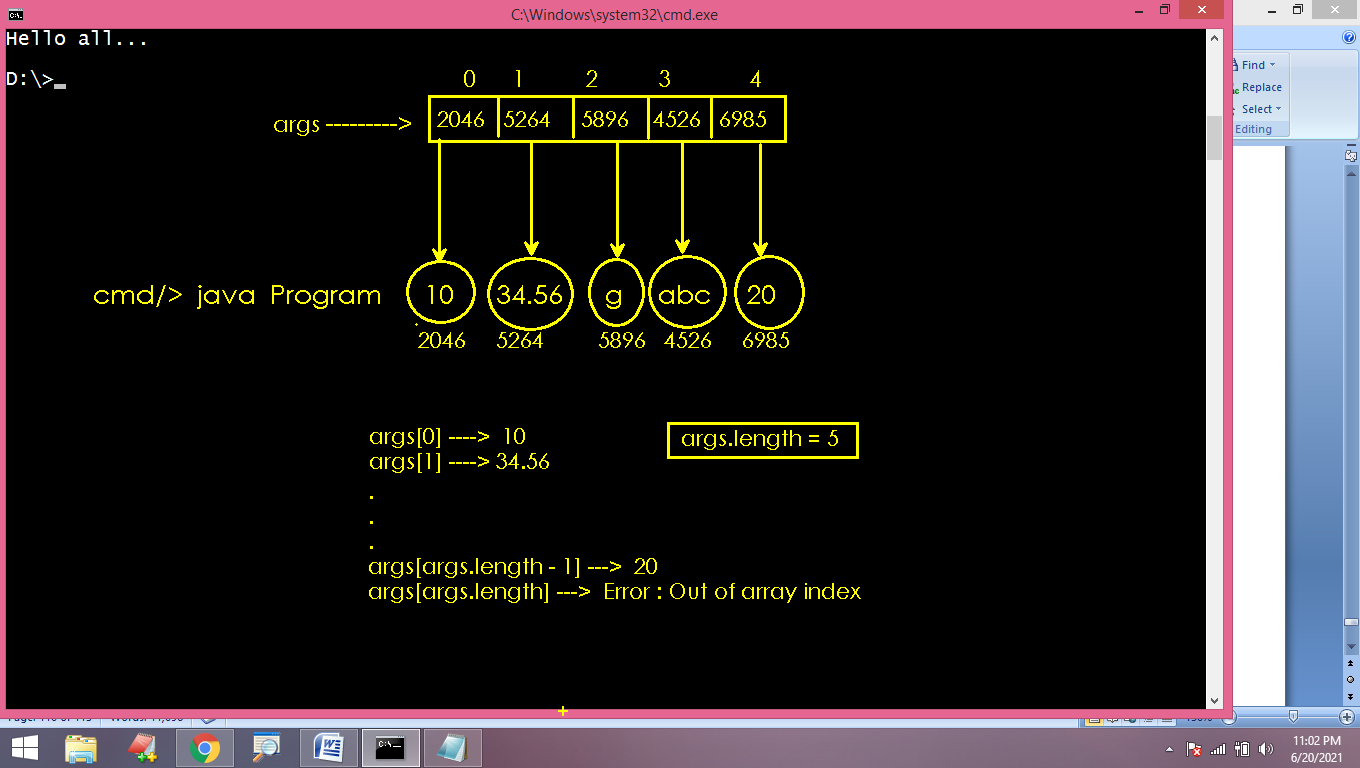
**Run java program from command line:**

* Open editor(notepad)
* Write your code
* Save with .java extension
* Open command prompt
* Move to source file location.
* Compile
  + Cmd/> javac file.java
* Run the code
  + Cmd/> java file
* You can see the output of code in command prompt



**Command line arguments:**

* “arguments” means “input” values
* We can pass input data from the command line while executing the program at command line.
* We can pass these input values followed by class name and by leaving the space between arguments.
* “main()” method collects all these arguments and store into String array type variable
  + main(String args[ ])
  + main(String[] args)
  + main(String arr[ ])
  + main(String[ ] arr)



class Demo

{

public static void main(String arr[ ])

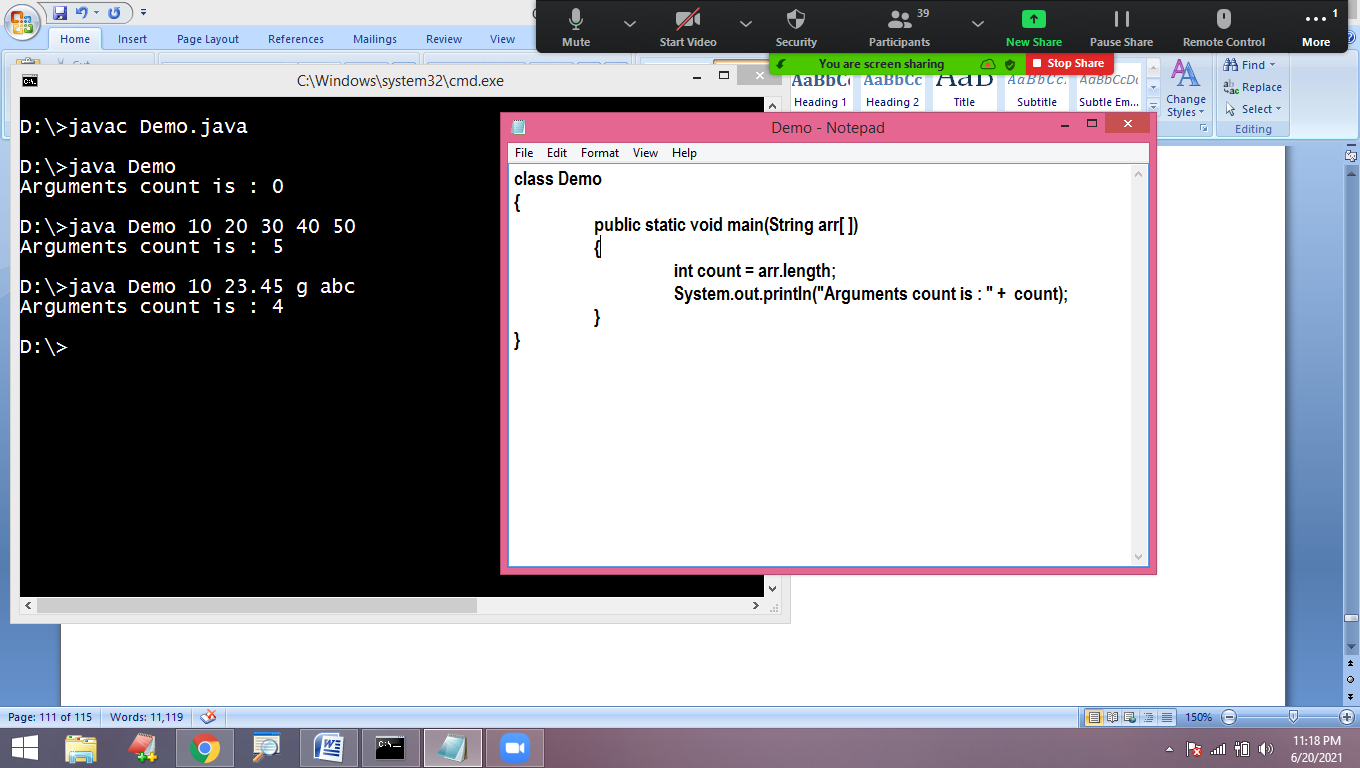
{

int count = arr.length;

System.out.println("Arguments count is : " + count);

}

}



class Demo

{

public static void main(String arr[ ])

{

int count = arr.length;

if(count==0)

{

System.out.println("No arguments");

}

else

{

System.out.println("List is : ");

for(int i=0 ; i<count ; i++)

{

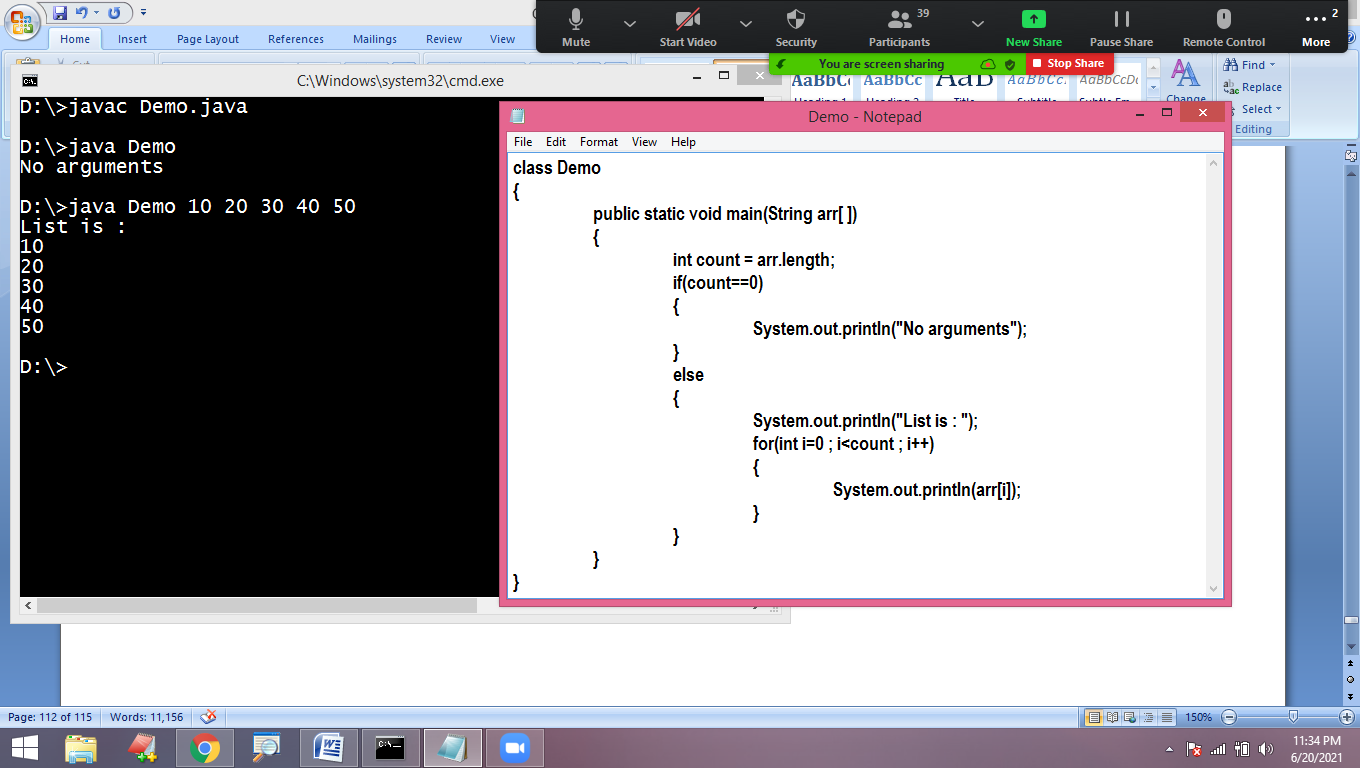
System.out.println(arr[i]);

}

}

}

}



* The arguments are String type by default.
* We need to convert these arguments into corresponding type using parse() methods of Wrapper classes.

class Demo

{

public static void main(String arr[ ])

{

int count=arr.length;

if(count<2)

{

System.out.println("Insufficient input values...");

}

else

{

int a = Integer.parseInt(arr[0]);

int b = Integer.parseInt(arr[1]);

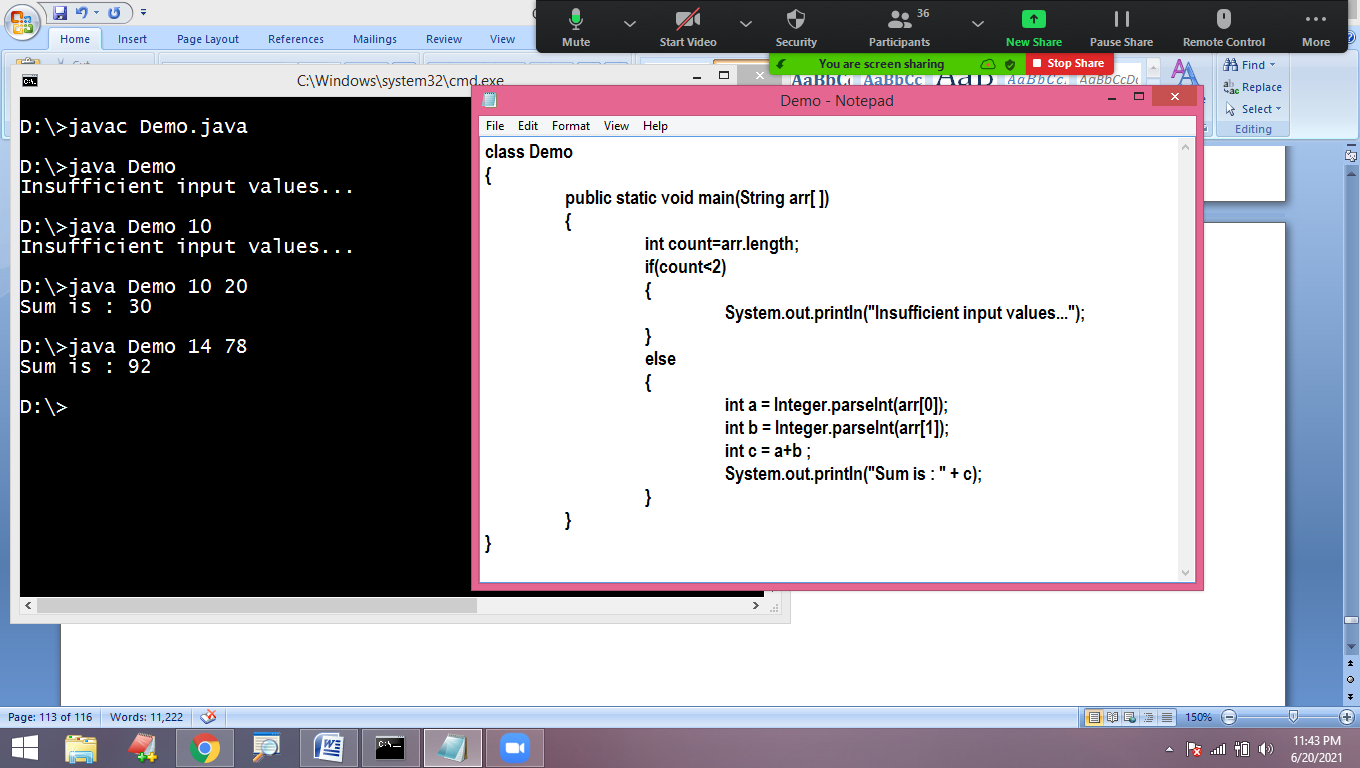
int c = a+b ;

System.out.println("Sum is : " + c);

}

}

}



Arrays

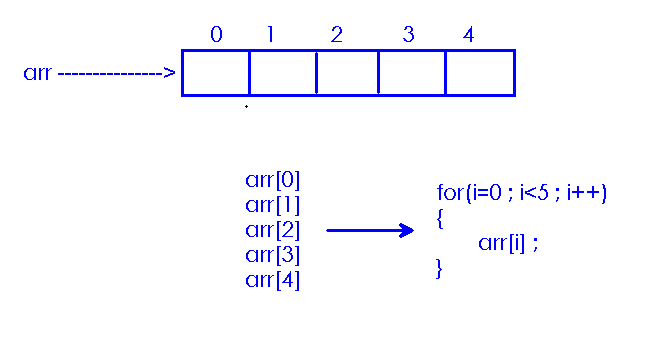
Array:

* Array is a collection of similar type of elements.
* In java, Array is an object.
* We can access array elements using their index starts with 0 to length-1
* We allocate memory to array in java using “new” operator.

Array creation:

data\_type name[ ] = new data\_type[size];

Note: It is always recommended to use loops to process array elements



Array automatically initializes with default values depends on its type:

**package** online;

**public** **class** Program1

{

**public** **static** **void** main(String[] args)

{

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

**boolean** arr[] = **new** **boolean**[5];

System.***out***.println("Array elements are : ");

**for**(**int** i=0 ; i<5 ; i++)

{

System.***out***.println(arr[i]);

}

}

}

Read the size of array and construct the Object:

**package** online;

**import** java.util.Scanner;

**public** **class** Program1

{

**public** **static** **void** main(String[] args)

{

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter size of array : ");

**int** n = scan.nextInt();

**int** arr[] = **new** **int**[n];

System.***out***.println("Array elements are : ");

**for**(**int** i=0 ; i<n ; i++)

{

System.***out***.println(arr[i]);

}

}

}

Read and store array elements:

**package** online;

**import** java.util.Scanner;

**public** **class** Program1

{

**public** **static** **void** main(String[] args)

{

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter size of array : ");

**int** n = scan.nextInt();

**int** arr[] = **new** **int**[n];

System.***out***.println("Enter array elements : ");

**for**(**int** i=0 ; i<n ; i++)

{

arr[i] = scan.nextInt();

}

System.***out***.println("Array elements are : ");

**for**(**int** i=0 ; i<n ; i++)

{

System.***out***.println(arr[i]);

}

}

}

We can directly initialize the array by assigning values using assignment operator:

**package** online;

**public** **class** Program1

{

**public** **static** **void** main(String[] args)

{

**int** arr[] = {10,20,30,40,50};

System.***out***.println("Array elements are : ");

**for**(**int** i=0 ; i<arr.length ; i++)

{

System.***out***.println(arr[i]);

}

}

}

Array is an Object in java. When we declare array variable, it is pointing to “null”. When we allocate memory then it starts pointing to memory location.

**package** online;

**public** **class** Program1

{

**static** **int** *arr*[] ;

**public** **static** **void** main(String[] args)

{

System.***out***.println("Default value : " + *arr*);

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

System.***out***.println("Memory location : " + *arr*);

}

}

for-each loop:

* It is called enhanced for loop.
* It is since jdk 1.5
* It is used to process the elements of Array or Collection easily.

Syntax :

for(datatype ele : array/collection)

{

Logic….

}

While displaying the elements of array using for each loop, no need to specify the bounds of array.

class Program

{

public static void main(String[] args)

{

int arr[ ] = {10, 20, 30, 40, 50};

System.out.println("Array elements are : ");

for(int ele : arr)

{

System.out.println(ele);

}

}

}

Advantage: We can simply work with array by without representing index. We can copy or display all elements of array easily.

Dis-advantage: We can move only in forward direction and access elements one by one only. If we want to display the elements in reverse order or to process only sub set of elements, we must use for-loop instead of for-each-loop.

class Program

{

public static void main(String[] args)

{

int arr[ ] = {10, 20, 30, 40, 50, 60, 70, 80};

for(int i=arr.length-1 ; i>=0 ; i--)

{

System.out.println(arr[i]);

}

System.out.println("----------");

for(int i=0 ; i<arr.length ; i=i+2)

{

System.out.println(arr[i]);

}

System.out.println("----------");

}

}

import java.util.Scanner;

class Program

{

static int arr[ ] ;

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter size of array : ");

int n = scan.nextInt();

arr = new int[n];

System.out.println("Enter array elements :");

for (int i=0 ; i<n ; i++)

{

arr[i] = scan.nextInt();

}

int sum=0;

for(int ele : arr)

{

sum = sum + ele ;

}

System.out.println("Sum of array elements : " + sum);

}

}

class Program

{

public static void main(String[] args)

{

int arr[ ] = {6, 3, 8, 2,9 ,1,5,7,4};

int ec=0, oc=0;

for(int ele : arr)

{

if(ele%2==0)

++ec;

else

++oc;

}

System.out.println("Even count is : " + ec);

System.out.println("Odd count is : " + oc);

}

}

class Program

{

public static void main(String[] args)

{

int arr[ ] = {6, 3, 8, 2,9 ,1,5,7,4};

int ec=0, oc=0;

for(int i=0 ; i<arr.length ; i++)

{

if(arr[i]%2==0)

++ec;

else

++oc;

}

System.out.println("Even count is : " + ec);

System.out.println("Odd count is : " + oc);

}

}

Find the largest element in the array :

class Program{ public static void main(String []args) { int arr[]={6,3,8,2,9,1,5,7,4}; int max=0,i=0; for(int ele:arr) { if(ele>max) max=ele; } System.out.println("maximum is:"+max); }}

public class Pgm3 { public static void main(String[] args) { int arr[] = {1, 6, 7, 3, 22, 27, 11}; int big = 0; for (int i = 0; i < arr.length; i++) { if (arr[i] > big) big = arr[i]; } System.out.println("largest no is: " + big); }}

Check the input number is duplicated or not:

import java.util.Scanner;

class Duplicate

{

public static void main(String[] args)

{

int count=0;

Scanner scan=new Scanner(System.in);

System.out.println("enter array size");

int n = scan.nextInt();

int arr[]=new int[n];

System.out.println("enter array elements : ");

for(int i=0;i<n;i++)

{

arr[i]=scan.nextInt();

}

System.out.println("enter a number to check :");

int ele = scan.nextInt();

for(int i=0;i<n;i++)

{

if(ele==arr[i])

{

count++;

}

}

if(count==0)

System.out.println("Element not present");

else if(count==1)

System.out.println("Not duplicated");

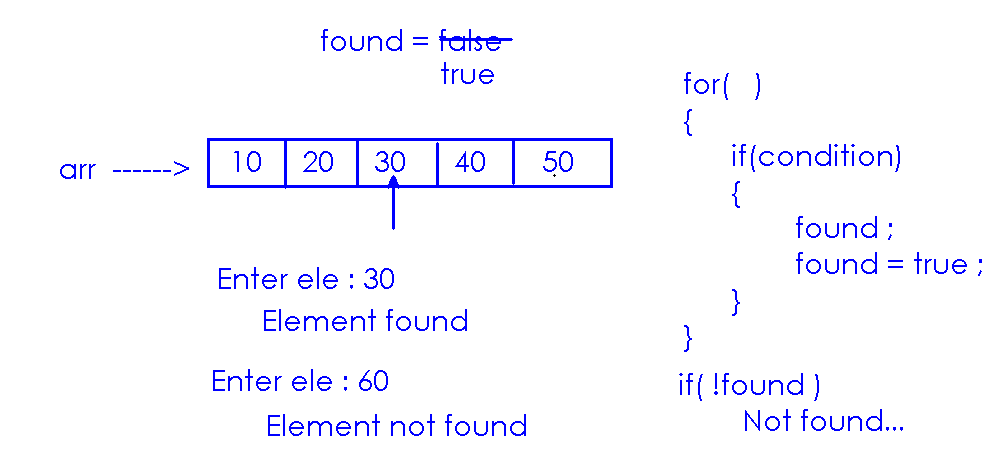
else

System.out.println("Duplicated");

}

}

Searching an element:



import java.util.\*;

public class Giridhar

{

public static void main(String[] args)

{

boolean found=false;

Scanner scan=new Scanner(System.in);

System.out.println("enter array size");

int n = scan.nextInt();

int arr[]=new int[n];

System.out.println("enter array elements");

for(int i=0;i<n;i++)

{

arr[i]=scan.nextInt();

}

System.out.println("enter a number");

int b = scan.nextInt();

for(int i=0;i<n;i++)

{

if(b==arr[i])

{

System.out.println("element found");

found=true;

break;

}

}

if(! found)

{

System.out.println("element not found");

}

}

}

Two dimensional array:

* Two dimensional array of elements are used to process the data in rows and columns.
  + datatype name[][] = new datatype[][];

class Demo

{

public static void main(String[] args)

{

int arr[ ][ ] = new int[3][3];

System.out.println("Length is : " + arr.length);

}

}

class Demo

{

public static void main(String[] args)

{

int arr[ ][ ] = new int[3][4];

int rows = arr.length;

int cols = arr[0].length;

System.out.println("Row length : " + rows);

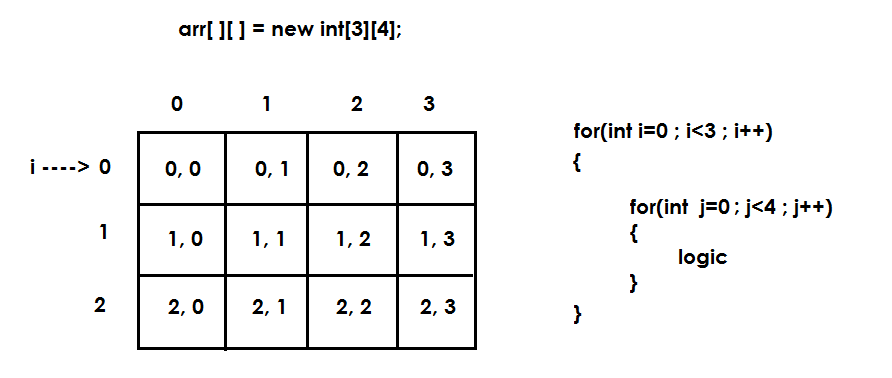
System.out.println("Column length : " + cols);

System.out.println("Length is : " + (rows\*cols));

}

}

* It is recommended to use nested for loop to process the elements of array.
* Outer loop represents number of rows.
* Inner loop represents number of columns.



class Demo

{

static int arr[ ][ ] = new int[3][4];

public static void main(String[] args)

{

System.out.println("Array elements are : ");

for (int i=0 ; i<3 ; i++)

{

for (int j=0 ; j<4 ; j++)

{

System.out.print(arr[i][j] + " ");

}

System.out.println();

}

}

}

Read elements into array:

import java.util.Scanner ;

class Demo

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter Dimension(row, column) : ");

int m = scan.nextInt();

int n = scan.nextInt();

int arr[][] = new int[m][n];

System.out.println("Enter " + (m\*n) + " elements into array : ");

for(int i=0 ; i<m ; i++)

for(int j=0 ; j<n ; j++)

arr[i][j] = scan.nextInt();

System.out.println("Array is : ");

for(int i=0 ; i<m ; i++)

{

for(int j=0 ; j<n ; j++)

{

System.out.print(arr[i][j] + "\t");

}

System.out.println();

}

}

}

**Object Oriented Programming**

**Introduction:**

* Java is Object oriented Programming Language.
* Java is **not** fully object oriented because it is supporting primitive types.
* **Wrapper classes** providing functionality to convert Primitive data into Objects in java applications.

**Object Oriented Programming Features:**

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

**Note:** Object oriented features neither belongs to java nor to any other programming language. These features are global. Every language can implement the functionality of Oriented Programming features. OOP functionality can be implemented using **‘Object and Class’.**

**Object: A real world entity consists**

1. **Identity:** Every object has unique identity. We use identities to call the objects.
2. **Variables/State:** Are used to store object information. Object information (state) is not constant.
3. **Methods/behavior:** Are used to process the information of object. Behavior can be used to change the State of Object.

**Object type:** Human

Identity: Amar

Properties: age, qualification, mobile\_no, mail\_id..…

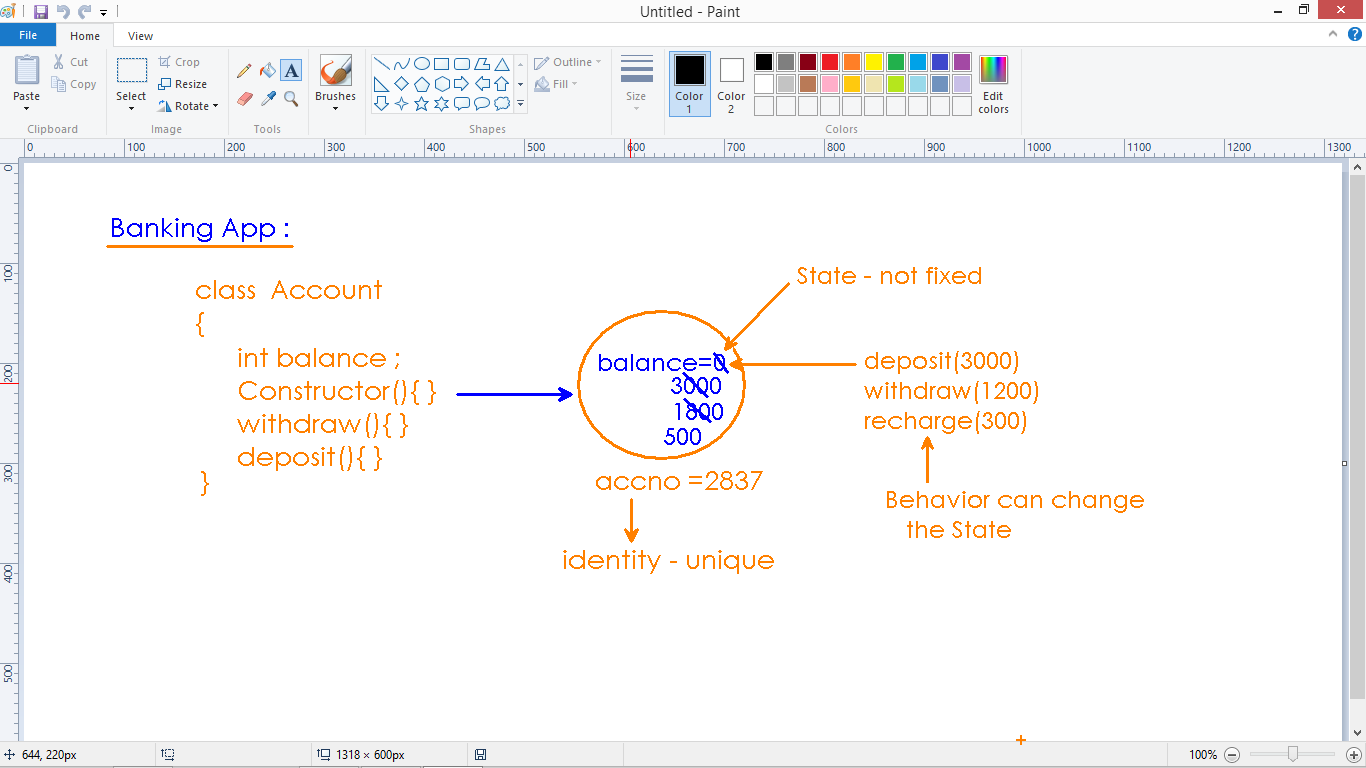
Functionalities: walk () , see() , run() , teach(), swim() , drive()....

**Object type:** Laptop

Identity: Lenovo

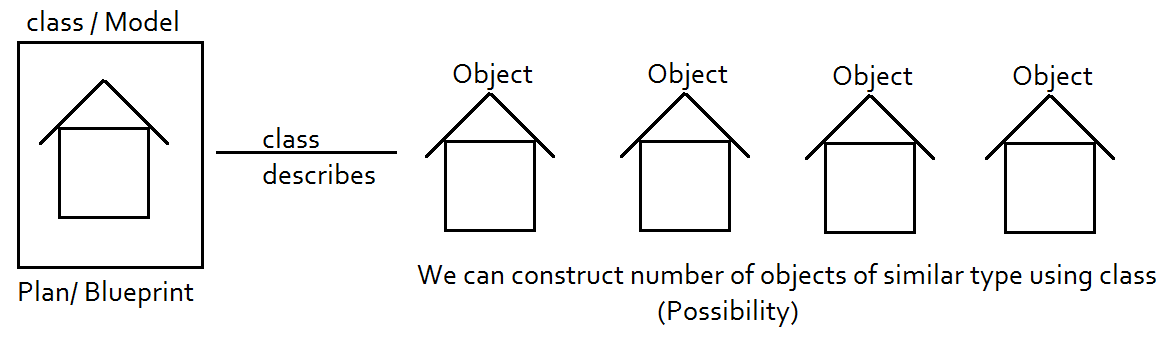
Properties: color, model, weight, configuration...

Functionalities: calculate () , play() , store() , send()....



**Class can be described as:**

* Class is a keyword
* Class is called user defined data type(extension to structure in C and C++)
* In Java application, every statement belongs to class.
* Class is the complete representation of Object.
* Without class definition, we cannot create object.



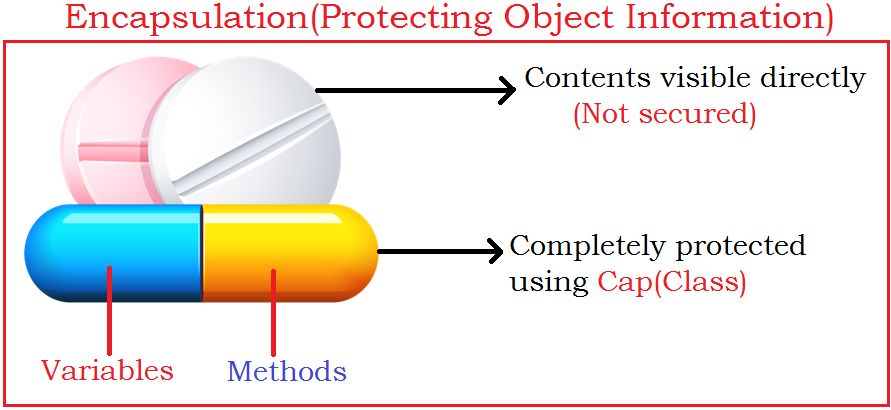
* We define objects in application for communication.
* In Communication process, different types of objects will participate.
* The functionality of these objects can be represented by different models(classes) as follows

1. POJO class
2. Bean class
3. Mutable class
4. Immutable class
5. Final class
6. Abstract class
7. Interface
8. Factory class
9. Singleton class
10. Exception class
11. Thread class
12. Generic class
13. Annotation

**Encapsulation:**

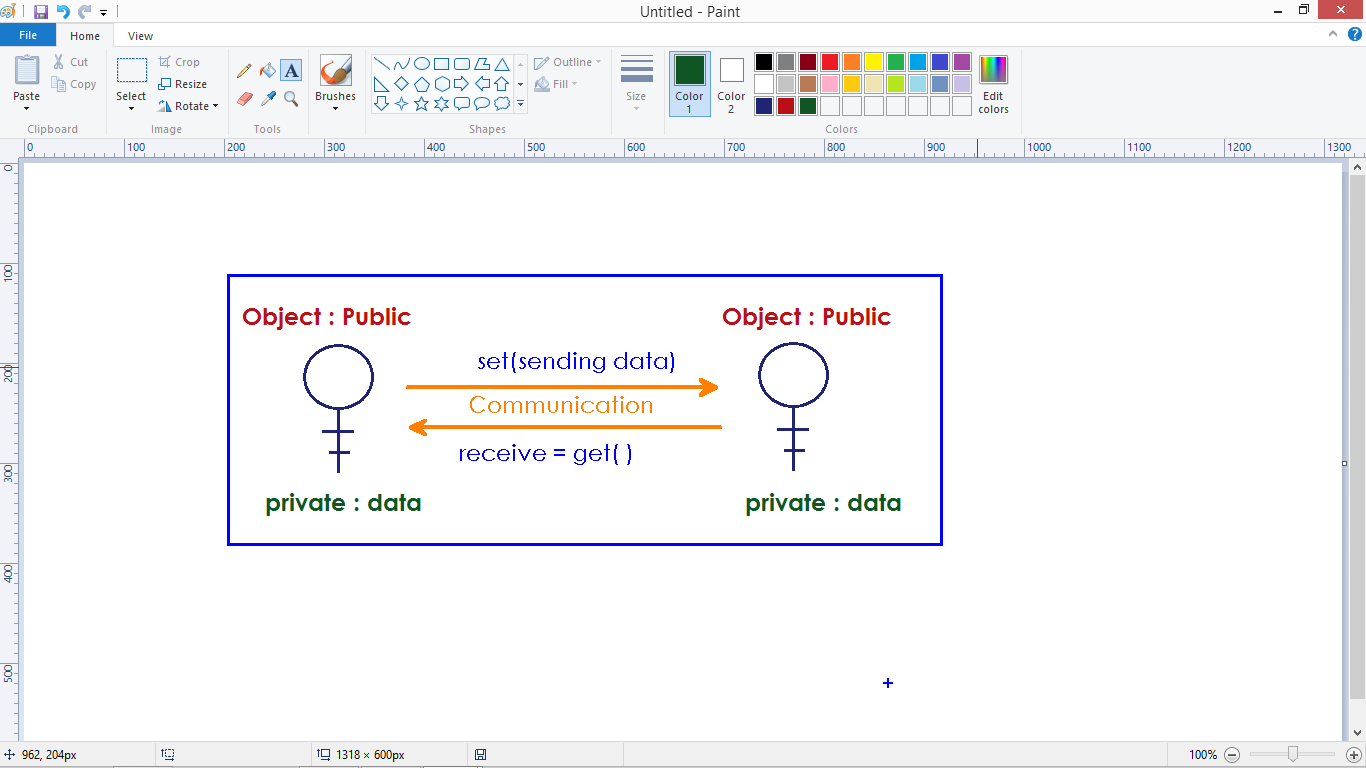
* Encapsulation is the concept of protecting object functionality from outside access.
* Writing data (variables) and code (methods) into a single unit.
* In the communication world, we should make outer visibility of Object but not properties and methods.

**Note:** Object information is sensitive and that has to be protected. Encapsulation is the concept of protecting object information. Class is used to implement encapsulation technically. POJO is the best example of how to protect information of a class.



**POJO:** Stands for **‘Plain Old Java Object’**. Every object is protected if it is defined as POJO. In java app, object should follow POJO rules.

1. **Class must be public:** Hence object is visible to all other object in communication process.
2. **Properties must be private:** Information is not visible to other object – hence objects cannot access the data directly.
3. **Every property has getter and setter:** Communication is possible using methods. Two standard methods get() and set() are given to send and receive the information.
4. ***Every class must have default constructor:*** Using default constructor, we can provide default values as object initialization. For example, balance in account is 0 as soon as account object has created.
5. **Can have optional arguments constructor:**  We can also set initial values instead of default values if required. For example, in some of the banks we can open account with minimum deposit amount.

****

**Getter method & Setter Method:**

* Object data cannot be accessed directly by another object.
* To get the information of an object, communication is required.
* Objects communication is possible using methods (setter and getter).
* Getter method returns info and doesn’t take input
* Setter method takes input but doesn’t return output

**For example:.**

private int balance ;

public int getBalance()

{

return this.balance;

}

public void setBalance(int balance)

{

this.balance = balance ;

}

**Simple POJO class:**

public class POJO

{

private int a, b;

public POJO()

{

// Initialize object with default values....

}

public POJO(int a, int b)

{

this.a = a;

this.b = b;

}

public int getA()

{

return this.a;

}

public int getB()

{

return this.b;

}

public void setA(int a)

{

this.a = a;

}

public void setB(int b)

{

this.b = b;

}

}

**Accessing POJO class:**

class Access

{

public static void main(String[] args)

{

POJO obj = new POJO();

System.out.println("a val : " + obj.getA());

System.out.println("b val : " + obj.getB());

}

}

class Access

{

public static void main(String[] args)

{

POJO obj = new POJO();

System.out.println("a val : " + obj.getA());

System.out.println("b val : " + obj.getB());

POJO obj1 = new POJO();

obj1.setA(10);

obj1.setB(20);

System.out.println("a val : " + obj1.getA());

System.out.println("b val : " + obj1.getB());

POJO obj2 = new POJO(30,40);

System.out.println("a val : " + obj2.getA());

System.out.println("b val : " + obj2.getB());

}

}

**When we apply the POJO rules to above class:**

public class Employee

{

private int num;

private String name;

private double salary;

public Employee()

{

// provide default values....

}

public Employee(int num, String name, double salary)

{

this.num = num ;

this.name = name ;

this.salary = salary ;

}

public int getNum()

{

return this.num;

}

public void setNum(int num)

{

this.num = num;

}

public String getName()

{

return this.name;

}

public void setName(String name)

{

this.name = name;

}

public double getSalary()

{

return this.salary;

}

public void setSalary(double salary)

{

this.salary = salary;

}

}

class AccessEmployeePOJO

{

public static void main(String[] args)

{

Employee e1 = new Employee();

// e1.eno = 101 ; /\*Error : Data cannot be accessed directly \*/

e1.setEno(101);

e1.setEname("Amar"); /\* Access data using functionality \*/

Employee e2 = new Employee(102 , "Annie");

System.out.println("e1 - ename : "+e1.getEname());

System.out.println("e2 - ename : "+e2.getEname());

}

}

**Default constructor:**

* Compiler adds a zero args constructor with empty body is called default constructor.
* We can check compiler added code using “javap” command

public class Test

{

public static void main(String[] args)

{

new Test(); // No error : It invokes default constructor

}

}

**Compiler doesn’t add default constructor when we define any constructor explicitly in class definition.**

public class Test

{

Test(int a)

{

// logic...

}

public static void main(String[] args)

{

new Test(); // Error : No such constructor

}

}

**We need to define all required constructors explicitly in the application:**

public class Demo

{

Demo()

{

// zero args...

}

Demo(int x)

{

// args constructor...

}

public static void main(String[] args)

{

new Demo();

}

}

**How can we access Non-static members in Java?**

**Answer:** Using Object address, we can access.

class Test

{

public static void main(String[] args)

{

Test obj = new Test();

obj.func();

Test.func(); // Error : cannot access using class name

}

void func()

{

System.out.println("Non static member");

}

}

**How to access static members in Java?**

Answer: Either by using class name or by using object address.

class Test

{

public static void main(String[] args)

{

Test.func();

Test obj = new Test();

obj.func();

}

static void func()

{

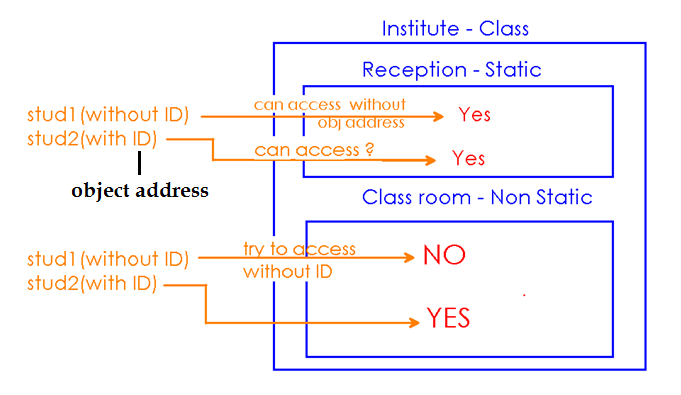
System.out.println("Non static member");

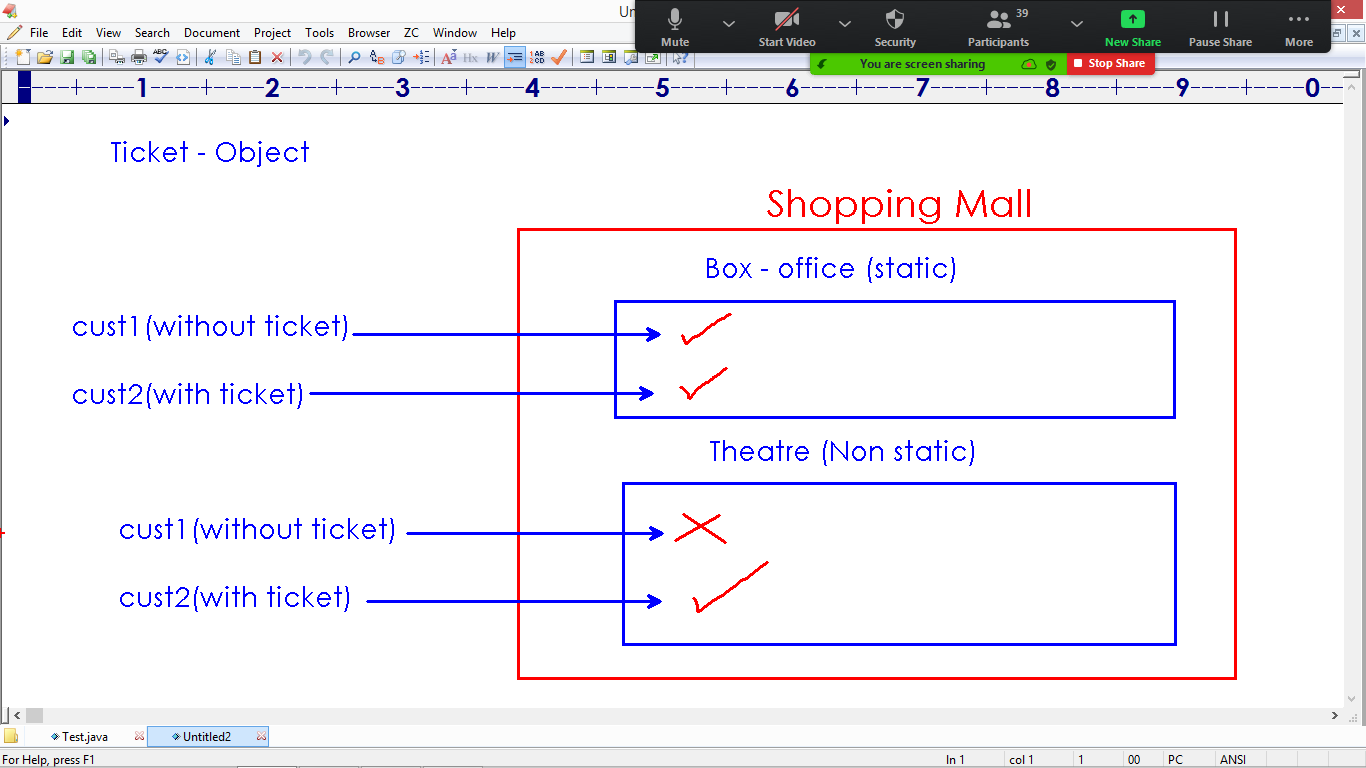
}

}

**Why it is allowed to access static members using Object address?**

We can access static members(shopping area) either by showing permission(ticket) or without permission but we can access Non static members(Theatre) only with permission(ticket).





**We can access both static and non static members using object address:**

class Test

{

static int x = 10;

int y;

public static void main(String[] args)

{

Test obj = new Test();

System.out.println("x val : " + obj.x);

System.out.println("y val : " + obj.y);

}

}

**Note:** In the above program, we can access both static and non static variables using object address.

**What if we have both static and non static with the same identity?**

* Static Member and Non static members cannot be defined with same identity.
* Every object is either static behavior or non static behavior with one identity.

class Pro

{

static int a=10;

int a ;

}

**Compile :** Error -> Variable 'a' is already defined.

**Why we cannot define both static and non static with same identity?**

* Every object is either static behavior (common data) or non static behavior(specific) with one identity.
* In the below program, we cannot define non-static variable bank\_name.
* In the same way, we cannot define variables accno, pin, name and balance as static.

class Account

{

// static variable holds common information of all Account Holders

static String bank\_name = "AXIS";

// instance variable holds specific information of object.

long accno ;

int pin ;

String name ;

int balance ;

}

**Static variable and Local variable can have the same name:**

class Pro

{

static int a ;

public static void main(String[] args)

{

int a = 10 ;

System.out.println("Staticvar : " + Pro.a);

System.out.println("Local var : " + a);

}

}

**Non static variable and Local variable can have the same name:**

class Pro

{

int a ;

public static void main(String[] args)

{

int a = 10 ;

System.out.println("Local var : " + a);

Pro obj = new Pro();

System.out.println("Instance var : " + obj.a);

}

}

**Constructor Chaining:**

* The concept of connecting multiple constructors in Object creation process.
* this() method is used to call constructor of same class explicitly.
* this() can access zero args constructor
* this(parameters) can access args constructor

**We can define more than one constructor with in the class:**

class Pro

{

Pro()

{

System.out.println("Zero args constructor");

}

Pro(int x)

{

System.out.println("Args constructor");

}

public static void main(String args[])

{

new Pro();

new Pro(10);

}

}

* In the above program, we created 2 objects to access 2 constructors.
* Using constructor chaining, we can access both the constructors in single object creation process.

**this():**

* It is used to access same class constructor.
* It must be used inside constructor.
* Call to this() must be the first statement in constructor.
* We can connect only one constructor from another constructor – hence it is called Chaining.

class Pro

{

Pro()

{

System.out.println("Zero args constructor");

}

Pro(int x)

{

this();

System.out.println("Args constructor");

}

public static void main(String args[])

{

new Pro(10);

}

}

**In constructor chaining, one constructor can be connected to only one constructor.Hence, Call to this() must be the first statement in constructor.**

class Pro

{

Pro()

{

System.out.println("Zero args constructor");

}

Pro(int x)

{

System.out.println("Args constructor");

this(); // Error : Must be the first statement

}

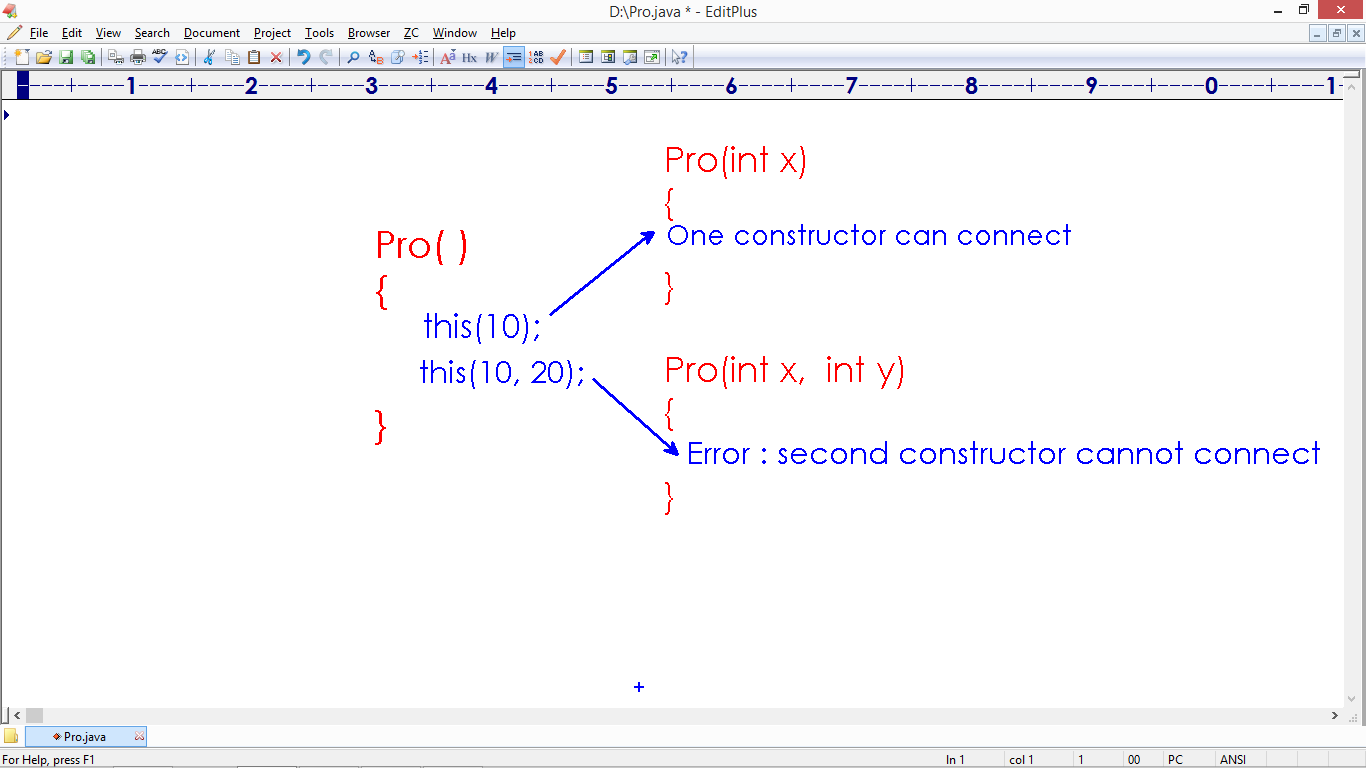
public static void main(String args[])

{

new Pro(10);

}

}



**Note:** We must use this() only inside another constructor of same class. Call to this(), must be the first statement in the constructor. Hence we can connect only one constructor from another constructor.

class Chain

{

Chain(int x, int y)

{

System.out.println("Two arguments...");

}

Chain()

{

this(10);

this(10,20); /\*Error : Call to this() must be the first statement\*/

System.out.println("Zero arguments...");

}

Chain(int x)

{

this(10,20);

System.out.println("One argument...");

}

public static void main(String[] args)

{

new Chain();

}

}

**Note:** We cannot call the constructor recursively

class Chain

{

Chain()

{

this(); // Error : recursive invocation

System.out.println("Zero args");

}

public static void main(String[] args)

{

new Chain();

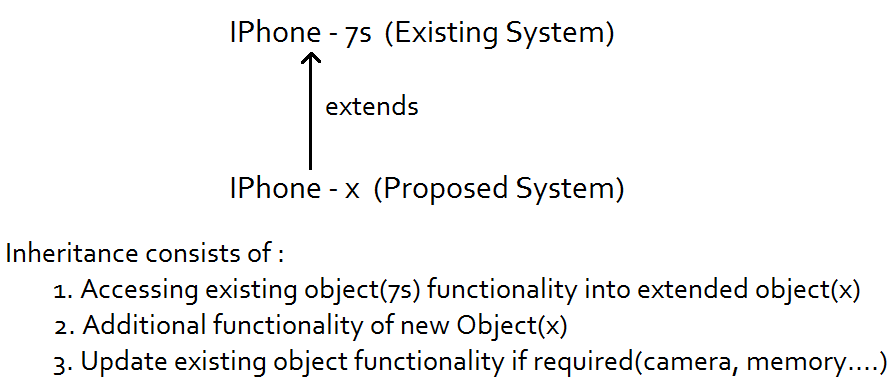
}

}

**Inheritance**

**Inheritance:**

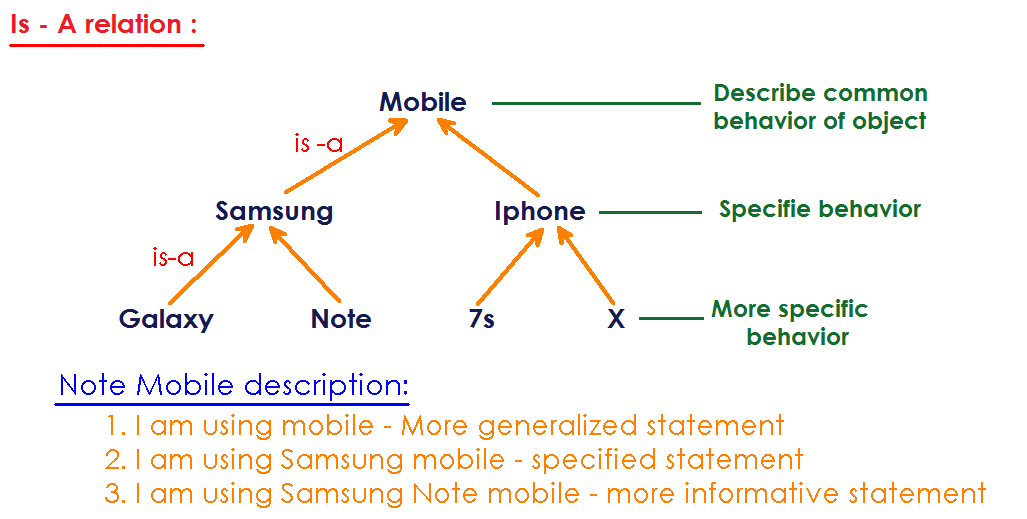
* Inheritance is the Concept of defining new Object by reusing the functionality of existing object.
* Create a class by using the functionality of existing class.
* Inheritance is used to update (release next version) of object.
* We can implement inheritance concept using ‘extends’ keyword in java.



**Inheritance** is also called ‘Is-A’ relation. The main advantage of Inheritance is code‘re**-usability’**.

**For example**

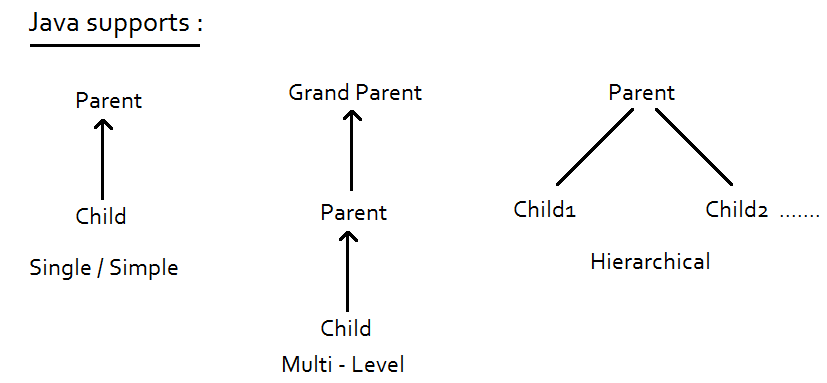
1. Swift "is-a" Car
2. Nokia "is- a" Mobile

****

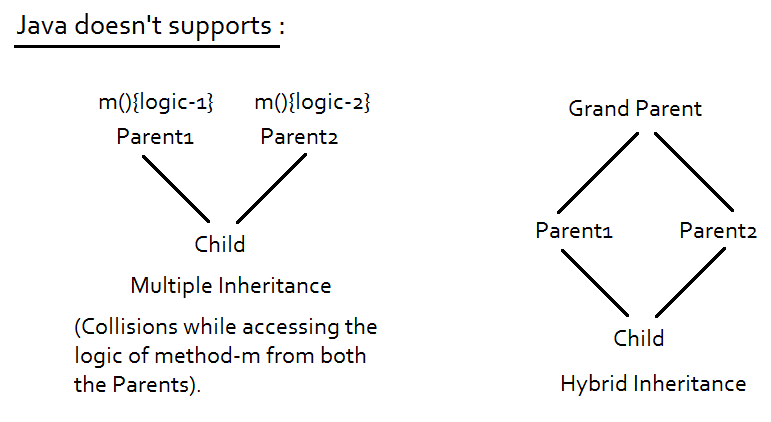
**Terminology:** After implementing "Is-A relation" between the Objects, we can call them in many ways…

* 1. Parent class - Child class
  2. Base class - Derived class
  3. Super class - Sub class
  4. Existing class – Extended class

**Types of inheritance:** Java supports only 3 types of Inheritance out of all the types supported by Object Oriented Programming.



* Java doesn’t support **Multiple inheritance and Hybrid inheritance**.
* Multiple inheritance is the concept of inherit the functionality of more than one class(chance of collisions if both the classes having same functionality).



**Single/Simple inheritance:** In Parent Child relation, we access the functionality Parent using Child but we cannot access Child functionality using Parent.

class Parent

{

static void m1()

{

System.out.println("Parent's m1");

}

}

class Child extends Parent

{

static void m2()

{

System.out.println("Child's m2");

}

}

class Single

{

public static void main(String[] args)

{

// Using Child, we can access both Parent and Child behavior

Child.m1();

Child.m2();

// Using Parent, we cannot access Child

Parent.m1();

// Parent.m2(); --> Error :

}

}

**Accessing Non static functionality:**

class JDK7

{

void features7(){

System.out.println("JDK 7 features...");

}

}

class JDK8 extends JDK7

{

void features8(){

System.out.println("JDK 8 features...");

}

}

class Inheritance

{

public static void main(String[] args)

{

JDK8 j8 = new JDK8();

j8.features7();

j8.features8();

JDK7 j7 = new JDK7();

j7.features7();

j7.features8(); // Error:

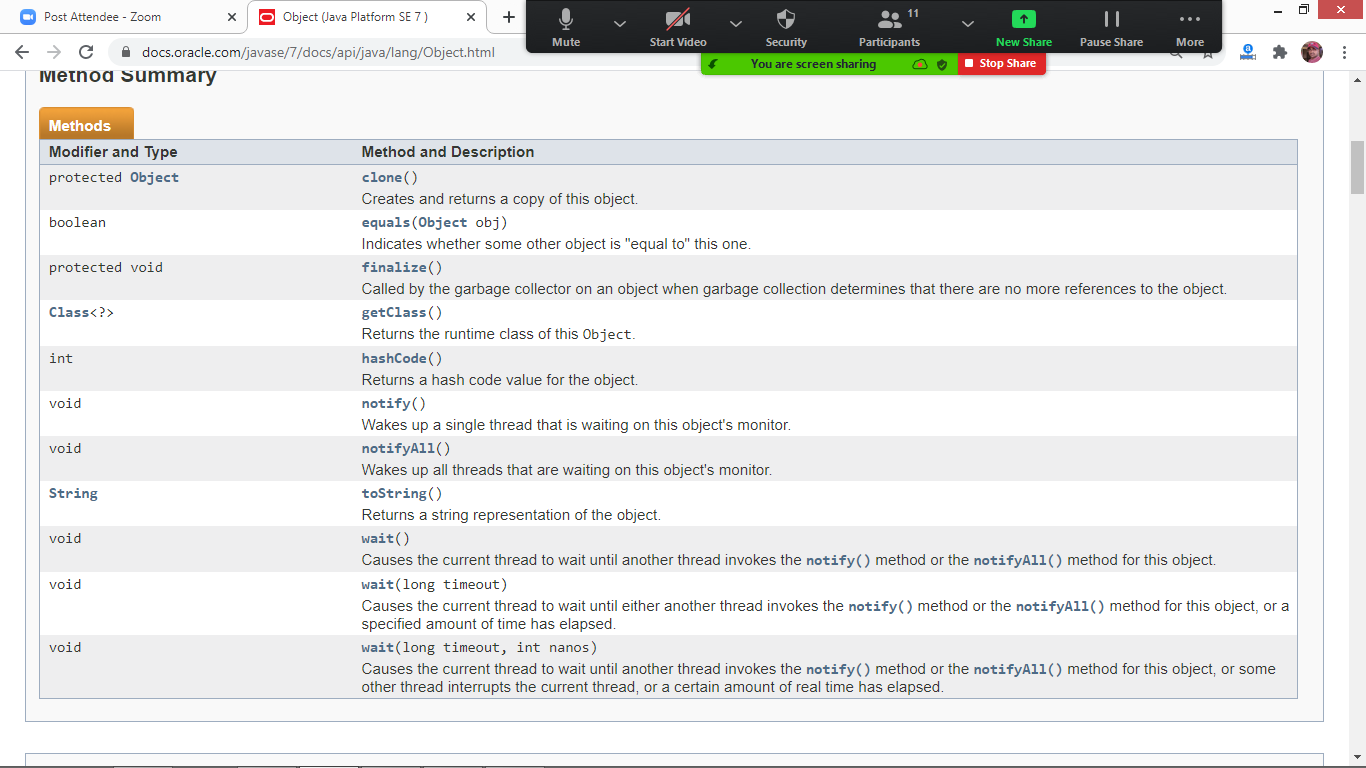
}

}

**What is the super class of all classes in java?**

* Object is pre-defined class belongs to ‘lang’ package.
* Object is the Parent class of all Java classes.
* Every class is implicitly inherits from Object class.
* As Object class is parent of all classes, we can access the functionality of Object class with every Custom or Other class object.

**Object class functionality as follows:**



**Note:** Every Java class shows Single inheritance behavior as it is extending from Object class.

class Custom

{

void fun()

{

System.out.println("Custom fun");

}

}

class Single

{

public static void main(String[] args)

{

Custom obj = new Custom();

obj.fun();

obj.hashCode(); // No error : Object class method

obj.getClass();

}

}

**How can we check the hashCode value of Object?**

* Every object gets memory in a random location.
* When we display address, the address will be displayed in hexadecimal format.
* hashCode is the integer value of memory address.
* We can check hashCode by calling the Object class method hashCode() on Object.

class Custom // extends Object

{

// body

}

class Single

{

public static void main(String[] args)

{

Custom obj = new Custom();

System.out.println("Custom address : " + obj.hashCode());

}

}

**Multi-Level inheritance:**

* Accessing class members in more than one level of hierarchy.
* Using child class object, it is possible to access the complete functionality of all the objects in the hierarchy.

/\*class Object(consider as Grand Parent)

{

int hashCode()

{

return object address in integer format.....

}

}\*/

class Parent // extends Object

{

void m1()

{

System.out.println("Parent class functionality");

}

}

class Child extends Parent

{

void m2()

{

System.out.println("Child class functionality");

}

}

class MultiLevelInheritance

{

public static void main(String[] args)

{

Child obj = new Child();

obj.m2();

obj.m1();

int address = obj.hashCode();

System.out.println("Object address : "+address);

}

}

* In the process of Child object creation, JVM instantiates Parent class first because we need to extend (update) Child from Parent only.
* As a programmer we can analyze whether the class is instantiating or not by defining a constructor.
* In java application it is not possible to create object without calling constructor.
* Hence Parent constructor invokes implicitly in the process of Child object creation.

class Parent

{

Parent()

{

System.out.println("Parent's object creation");

}

}

class Child extends Parent

{

Child()

{

System.out.println("Child's object creation");

}

}

class Check

{

public static void main(String[] args)

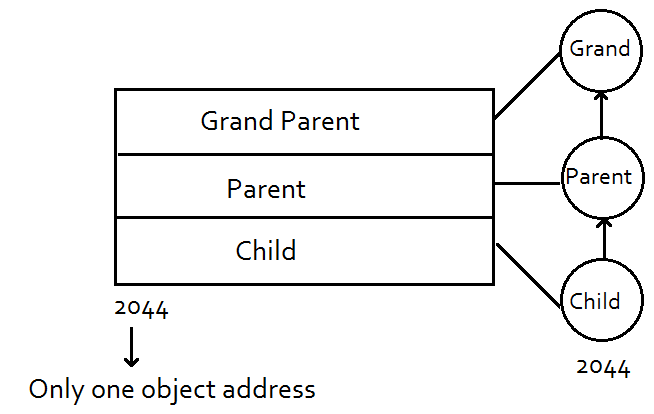
{

new Child();

}

}

* JVM doesn’t allocate memory in different locations for different objects in the hierarchy.
* The complete functionality of all the objects in the hierarchy can be accessed only by using only one object address (that is Child class).



**Following program prints same address for both Parent and Child:**

class Grand

{

Grand()

{

System.out.println("Grand's : " + this);

}

}

class Parent extends Grand

{

Parent()

{

System.out.println("Parent's : " + this);

}

}

class Child extends Parent

{

Child()

{

System.out.println("Child's : " + this);

}

}

class Check

{

public static void main(String[] args)

{

new Child();

}

}

**Output:**

Grand's : Child@106d69c

Parent's : Child@106d69c

Child's : Child@106d69c

class Samsung

{

Samsung(){

System.out.println("Samsung features");

System.out.println(this);

}

}

class Guru extends Samsung{

Guru(){

System.out.println("Guru features");

System.out.println(this);

}

}

class Galaxy extends Guru{

Galaxy(){

System.out.println("Galaxy features");

System.out.println(this);

}

}

class Inheritance {

public static void main(String[] args) {

Galaxy obj = new Galaxy();

}

}

**Method Overriding:** Defining a method in Child with the same name and signature of its Parent class.

class Parent

{

void fun()

{

System.out.println("Parent's functionality");

}

}

class Child extends Parent

{

void fun() // Method overriding

{

System.out.println("Child's functionality");

}

public static void main(String[] args)

{

Child obj = new Child();

obj.fun(); // It is looking for the method in Child class first. If it is not present in Child then it searches in Parent class.

}

}

**Advantage of Method overriding:** Overriding is the concept of updating the functionality of Parent object from Child. We override the existing functionality of Parent class Object in Child class when it is not sufficient to Child. It is the part of inheritance concept.

class Galaxy

{

camera()

{

...... 8 megapixel

}

}

class Note extends Galaxy

{

camera()

{

...... 16 megapixel

}

}

**Inheritance mechanism includes:**

* Access Parent object functionality
* Adding new functionality in Child
* Update(override) Parent object functionality if required.

class SamsungGuru

{

void call()

{

System.out.println("Guru - Call");

}

void camera()

{

System.out.println("Guru - Camera - 2MP");

}

}

class SamsungGalaxy extends SamsungGuru

{

void videoCall()

{

System.out.println("Galaxy - VideoCall");

}

void camera() // override - rewrite - update

{

System.out.println("Galaxy - Camera - 8MP");

}

}

class Inheritance

{

public static void main(String[] args)

{

SamsungGalaxy g1 = new SamsungGalaxy();

g1.call(); // Accessing existing object feature

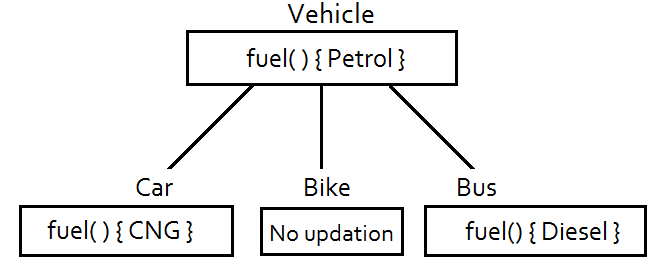
g1.videoCall(); // New feature

g1.camera(); // Updated feature

}

}

**Hierarchical Inheritance:** Sharing the properties of Object to multiple child objects.



class Vehicle

{

String fuel()

{

return "Petrol";

}

}

class Car extends Vehicle

{

String fuel()

{

return "CNG";

}

}

class Bike extends Vehicle

{

// no updation

}

class Bus extends Vehicle

{

String fuel()

{

return "Diesel";

}

}

class Inheritance

{

public static void main(String[] args)

{

Car c = new Car();

Bike b = new Bike();

Bus bb = new Bus();

System.out.println("Car fuel type : " + c.fuel());

System.out.println("Bike fuel type : " + b.fuel());

System.out.println("Bus fuel type : " + bb.fuel());

}

}

**Accessing overridden method of Parent class:**

class Parent

{

void fun()

{

System.out.println("Parent's fun");

}

}

class Child extends Parent

{

void fun()

{

System.out.println("Child's fun");

}

}

class Check

{

public static void main(String[] args)

{

Child c = new Child();

c.fun();

Parent p = new Parent(); **// It is not recommended**

p.fun();

}

}

**Note:** In the above application, we create Parent class object separately to call Parent class functionality. In Child object creation, we get access of Parent class. Create duplicate object to Parent class wasting the memory. It is not recommended.

**Solution:** to the above problem is using “super” keyword to access Parent functionality from Child class.

**super:**

|  |  |
| --- | --- |
| **this** | **super** |
| 1. It is a keyword | 1. It is a keyword |
| 1. It is a pre-defined non static variable | 1. It is a pre-defined non static variable |
| 1. It is used to access current object(class) functionality | 1. It is used to access Parent object(class) functionality from the Child |
| 1. It must be used only in non-static context | 1. It must be used only in non-static context |
| 1. It holds object address | 1. **It doesn’t hold any object address** |

class Parent

{

void fun()

{

System.out.println("Parent's fun");

}

}

class Child extends Parent

{

void fun()

{

System.out.println("Child's fun");

}

void access() // non static area

{

this.fun(); // access Child class fun

super.fun(); // access Parent class fun

}

}

class Check

{

public static void main(String[] args)

{

Child c = new Child();

c.access();

}

}

**Super doesn’t holds Parent object address:**

class Parent

{

// empty

}

class Child extends Parent

{

Child()

{

System.out.println("Child's address : " + this);

// System.out.println("Parent's address : " + super); // Error : 'super' doesn't holds address

}

}

class Check

{

public static void main(String[] args)

{

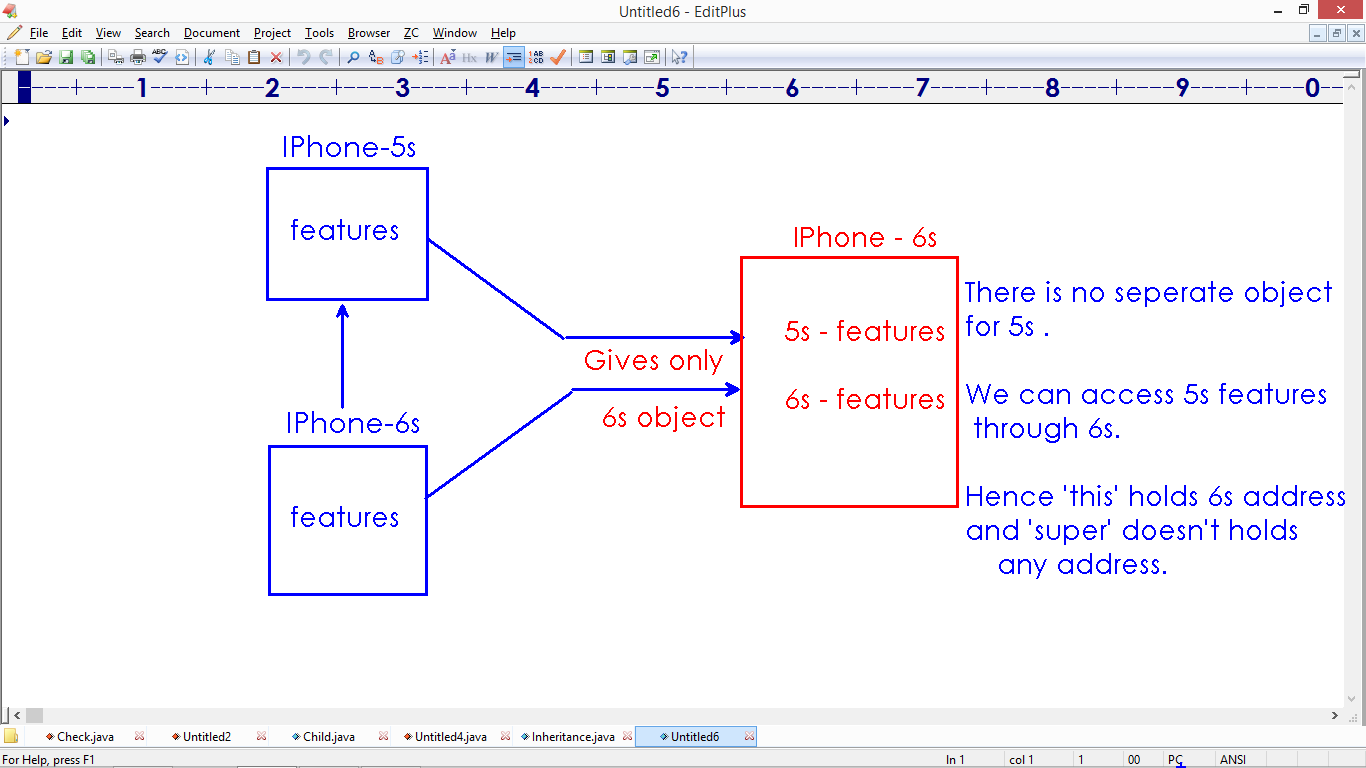
new Child();

}

}

**Why ‘super’ doesn’t holds Parent address?**

* We can access the complete functionality of all the objects in the hierarchy using single address (Child’s).
* There is no separate Parent object to hold the address by super keyword.
* We can say ‘super’ is the sub pointer to ‘this’ keyword and it is used to access Parent functionality from Child.



**super():**

* super() is used to access Parent class constructor from the child class. It is used to initialize Parent object from Child.
* We must use super() method inside the child class constructor only.
* Call to super() method must be the first statement in the Child’s constructor. Calling(connecting) Parent constructor from Child is called ‘Constructor Chaining’ in Is-A relation.
* In the process of child class object construction, it is not possible to invoke parent’s constructor explicitly to initialize non static variables of Parent class.

|  |  |
| --- | --- |
| **this()** | **super()** |
| 1. It is used to invoke current class constructor. | 1. It is used to invoke Parent class constructor from Child. |
| 1. It must be used only inside another constructor of same class. | 1. It must be used only inside Child’s constructor. |
| 1. Call to this() must be the first statement. | 1. Call to super() must be the first statement. |
| 1. To connect constructors in a single class. | 1. To connect constructors in Is-A relation. |
| 1. Used to initialize an object via multiple constructors. | 1. **Used to initialize parent class object in the process of child object creation.** |

**Note:** In the process of child class object construction, JVM implicitly creates Parent’s object first. For this construction, JVM uses super() method.

/\*class Object

{

Object()

{

Object creation.....

}

}\*/

class Parent extends Object

{

Parent()

{

// super(); /\* compiler added code \*/

System.out.println("Parent's instantiation.....");

}

}

class Child extends Parent

{

Child()

{

// super(); /\* compiler added code \*/

System.out.println("Child's instantiation.....");

}

public static void main(String[] args)

{

Child c = new Child();

}

}

**When we call super() method explicitly?**

* In the process of Child object creation it is not possible to invoke Parent class constructor explicitly to provide initialization.
* We use super() method in the child class constructor to access Parent’s constructor. Hence we can initialize Parent object.

class Parent

{

int a ;

Parent(int a)

{

this.a = a ;

}

}

class Child extends Parent

{

Child(int a)

{

super(a); /\* calling Parent's arguments constructor \*/

}

}

class MainClass

{

public static void main(String[] args)

{

Child obj = new Child(10);

}

}

**Initializing Parent and Child objects:**

class Parent

{

int a ;

Parent(int a)

{

this.a = a ;

}

}

class Child extends Parent

{

int b ;

Child(int a, int b)

{

super(a);

this.b = b ;

}

void display()

{

System.out.println("Parent's a value : "+super.a);

System.out.println("Child's b value : "+this.b);

}

}

class MainClass

{

public static void main(String[] args)

{

Child obj = new Child(10,20);

obj.display();

}

}

**Practice this code:**

class Parent extends Object

{

int a, b ;

Parent(int a, int b)

{

this.a = a ;

this.b = b ;

}

}

class Child extends Parent

{

int x, y;

Child(int a, int b, int x, int y)

{

super(a, b);

this.x = x ;

this.y = y ;

}

void details()

{

System.out.println("x value : " + this.x);

System.out.println("y value : " + this.y);

System.out.println("a value : " + super.a);

System.out.println("b value : " + super.b);

}

}

class Inheritance

{

public static void main(String[] args)

{

Child obj = new Child(10, 20, 30, 40);

obj.details();

}

}

**Note:** Compiler writes a default constructor only when we don’t define any constructor in the class. Hence we need to check the entire constructor is available while connecting them in Constructor Chaining process.

**Code-1:**

class Parent

{

}

class Child extends Parent

{

}

class Inheritance

{

public static void main(String[] args)

{

new Child(); // invoke Default constructors of Parent and Child to create object.

}

}

**When we don’t define any constructor, then only compiler will add default constructor.**

**The code as follows:**

class Parent

{

/\*Parent()

{

}\*/

}

class Child extends Parent

{

/\*Child()

{

super();

}\*/

}

class Inheritance

{

public static void main(String[] args)

{

new Child();

}

}

**If we write any arguments constructor, compiler will not add default constructor(zero args).**

class Parent

{

Parent(int a)

{

}

}

class Child extends Parent

{

/\*Child()

{

super(); // Error : trying to invoke default constructor implicitly.

}\*/

}

class Inheritance

{

public static void main(String[] args)

{

new Child();

}

}

class Parent

{

Parent()

{

System.out.println("Parent's zero args constructor");

}

Parent(int a)

{

System.out.println("Parent's args constructor");

}

}

class Child extends Parent

{

Child()

{

// super();

System.out.println("Child's zero args constructor");

}

Child(int a)

{

System.out.println("Child's args constructor");

}

}

class Inheritance

{

public static void main(String[] args)

{

new Child(); // Create objects with zero args constructors

}

}

* We can define all the required constructors explicitly in the program and these constructors can be connected using this() and super() in the process of constructor chaining.
* We can connect only one constructor from another constructor because call to either this() or super() method must be the first statement in the constructor.

class Parent

{

Parent()

{

this(10);

System.out.println("Parent's zero args constructor");

}

Parent(int a)

{

System.out.println("Parent's args constructor");

}

}

class Child extends Parent

{

Child()

{

this(10);

System.out.println("Child's zero args constructor");

}

Child(int a)

{

super();

System.out.println("Child's args constructor");

}

}

class Inheritance

{

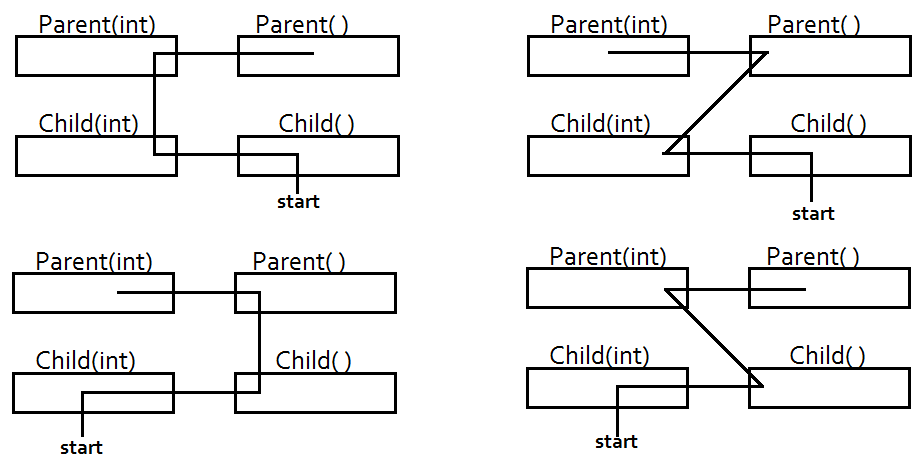
public static void main(String[] args)

{

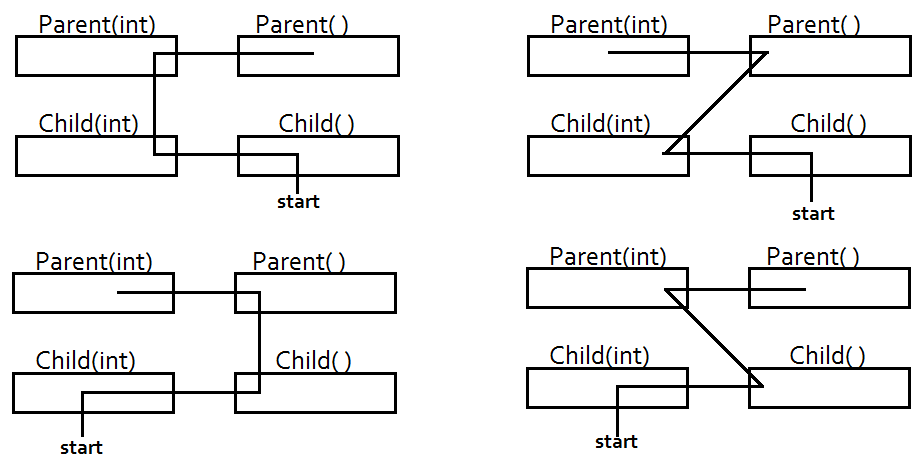
new Child();

}

}



**Write code to connect the constructors in following order.**

****

class Parent

{

Parent()

{

System.out.println("Parent's zero args constructor");

}

Parent(int a)

{

this();

System.out.println("Parent's args constructor");

}

}

class Child extends Parent

{

Child()

{

super(10);

System.out.println("Child's zero args constructor");

}

Child(int a)

{

this();

System.out.println("Child's args constructor");

}

}

class Inheritance

{

public static void main(String[] args)

{

new Child(10);

}

}

* Call to super() or this() must be the first statement in the constructor.
* This is why we can connect only one constructor from another.

class Parent

{

Parent()

{

}

}

class Child extends Parent

{

Child()

{

super();

this(10); // not allowed - must be the first statement

}

Child(int a)

{

}

}

class Inheritance

{

public static void main(String[] args)

{

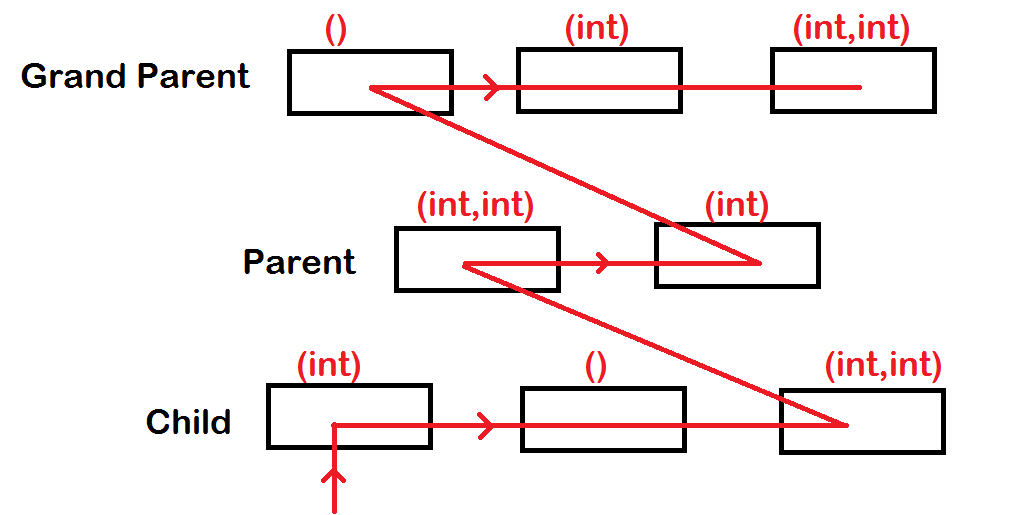
new Child();

}

}

**How to connect constructors in Multi level inheritance:**

In the process of connecting constructors, first we need to connect all the constructors in Child level before we are moving to Parent level. We can connect only one constructor at a time.

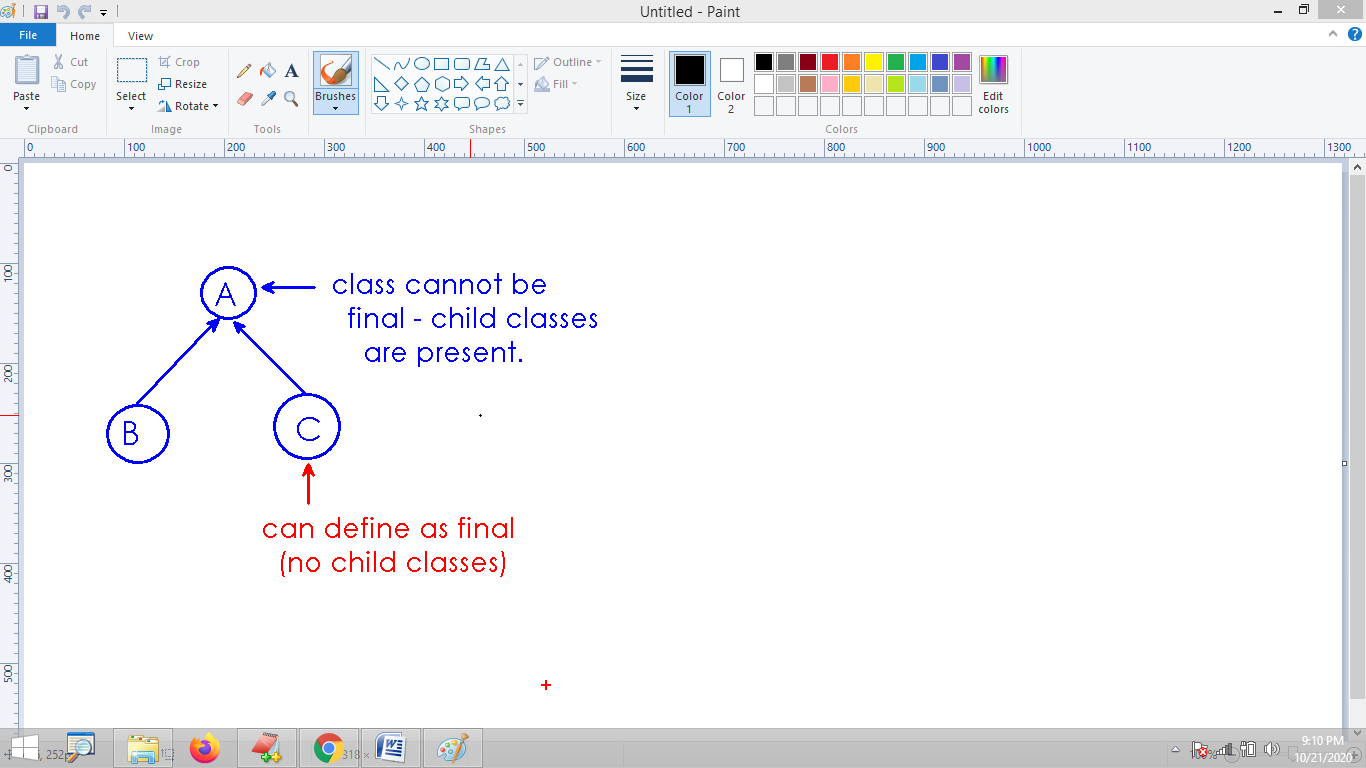


**final:**

* It is a keyword or modifier.
* Final member is constant, hence cannot be modified.
* Final is used to set restrictions on object updations.
* We can apply final to class, variable and method.

**Class is final:**

* Class represents Object.
* If we define class as final that cannot be inherited (extended).
* Only leaf classes (no extensions) in the hierarchy recommended defining as “final”.



final class A

{

// logic

}

class B extends A /\* Error : final class-A cannot be extended \*/

{

// logic

}

**Method is final:**

* If method is final, that cannot be overridden.
* Overriding is the concept of updating Parent functionality into Child.
* Final methods of Parent class cannot be updated from Child class.
* For example if we want to restrict only a specific method not to be updated in the class can be defined as final.

class Galaxy

{

final void call(){

// logic......

}

void camera(){

// 12 mp .....

}

}

class Edge extends Galaxy

{

void call(){

// Error : cannot override final method

}

void camera() {

// 16 mp .....

}

}

**Variable is final:**

* Variable holds data.
* If variable is final that cannot be modified.
* Constant variables in java application must be defined as final.
* Most of the final variables are static in java application (because constant variables are common for all), for example….
  + **static final double PI = 3.142 ;**

class Test

{

static final double PI = 3.142 ;

public static void main(String[] args)

{

Test.PI = 3.1412 ; // Error : can't modify

}

}

**Note:** We cannot update the functionality of “final object”, but we can access the complete functionality of Object.

final class FinalClass

{

static final int a = 10 ;

final void fun()

{

System.out.println("Final method....");

}

}

class AccessFinalClass

{

public static void main(String[] args)

{

System.out.println("Final class a value : "+FinalClass.a);

FinalClass obj = new FinalClass();

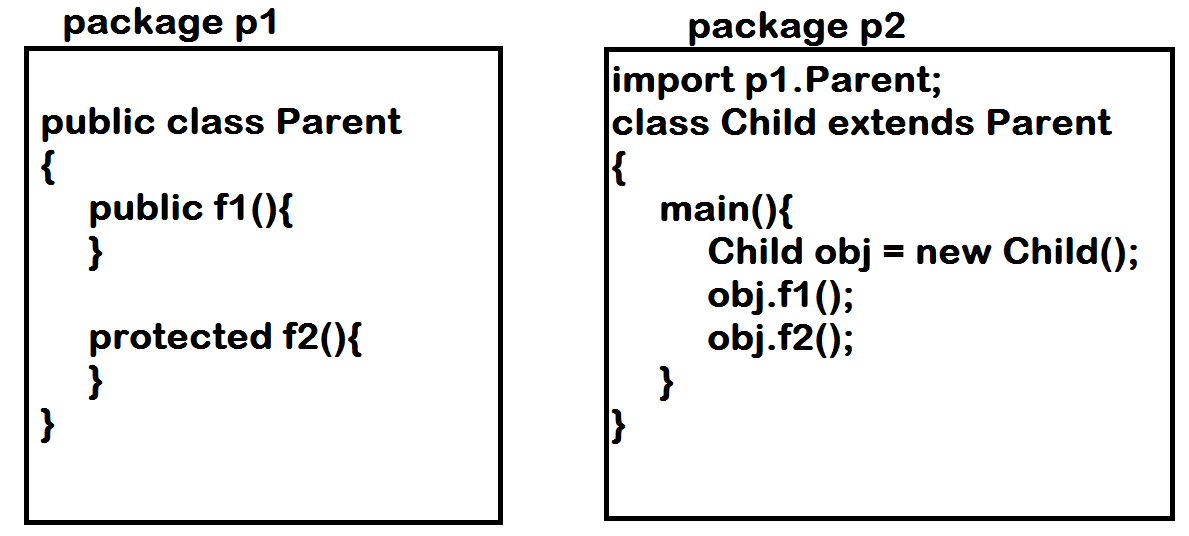
obj.fun();

}

}

**Accessing protected members of super class:**

* Protected members of Parent Object must be done through Child Object.
* By just creating Object for the Parent class from the class, it is not possible to access protected members.
* JVM cannot understand the relation between Super and Sub Object if we create only Object of Super class.



package p1;

public class Parent

{

private void m1(){

}

void m2(){

}

protected void m3(){

}

public void m4(){

}

}

package p2;

import p1.Parent;

public class Child extends Parent

{

public static void main(String[] args)

{

Child obj = new Child();

obj.m4();

obj.m3();

obj.m2(); // Error : package level - cannot access from another package.

obj.m1(); // Error : private access

}

}

**Can we override the constructor?**

* Constructor cannot be overridden.
* Constructor definition belongs to a Particular class.
* Constructors name same as Class name. Hence we cannot override the Parent class constructor into Child class.

class Parent

{

Parent()

{

/\* logic \*/

}

}

class Child extends Parent

{

Parent()

{

/\* Error : Class name and Constructor name mismatch \*/

}

}

**Can we apply “final” modifier to constructor?**

* We can’t apply final modifier to constructor.
* final modifier restricts updations.
* As we are not updating(override) constructor anyway, no need to apply final modifier to constructor.

class Test

{

final Test()

{

// Error : we cannot apply final modifier to constructor.

}

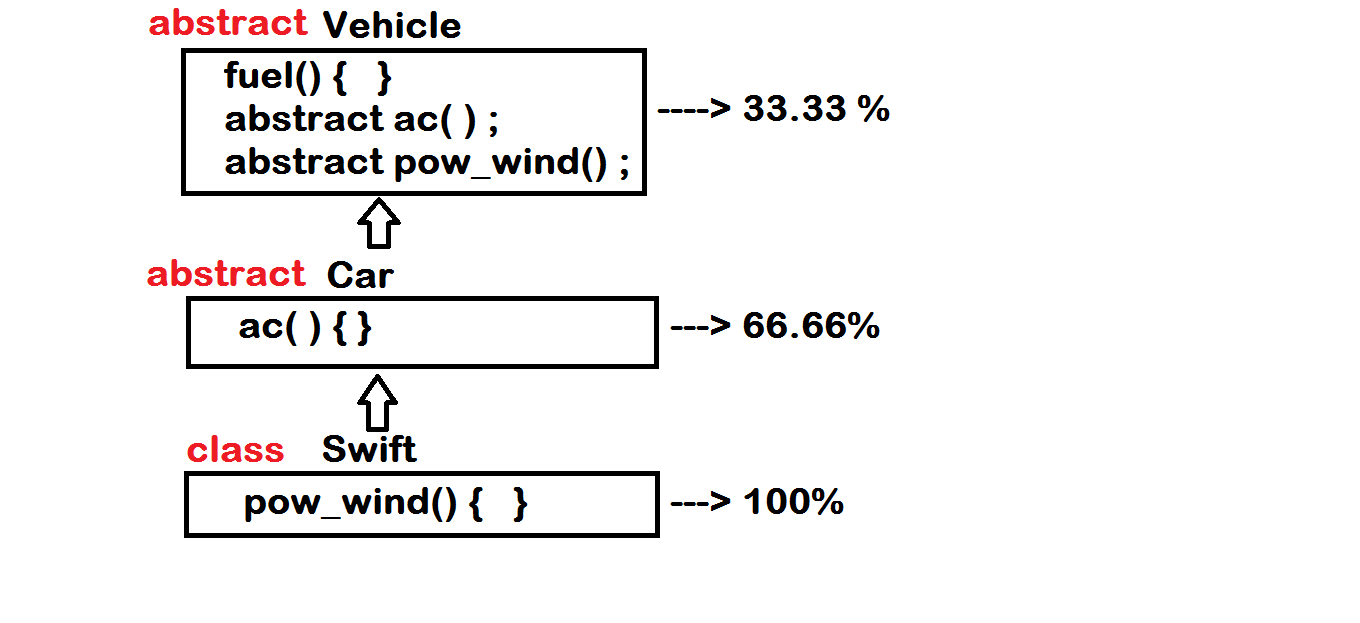
}

**Abstraction**

* Abstraction is the concept of hiding unnecessary details of Object and shows only essential features.
* Abstraction provides the information about "What object can do instead how it does it".
* Abstraction is the "General View" of Object.

**Abstract class:**

* If a class is not able to provide complete definition of Object referred as Abstract class.
* The class which is not providing complete definition of Object.
* "abstract" is a pre-defined keyword allowed to define abstract classes.
* Abstract class also saved with “.java” extension only.
* “.class” file will be generated when abstract class has compiled.



* “abstract” modifier is used to define abstract classes.
* Abstract class is allowed to define
  + Abstract methods(methods don’t have body)
  + Concrete methods(methods having body)

abstract class AbstractClass

{

// abstract class contains

void concreteMethod()

{

// empty

}

abstract void abstractMethod() ;

}

* Abstract class cannot be instantiated directly using “new” keyword, because it was not fully defined.
* For example, we can release any Mobile into the market after 100% manufactured.

abstract class AbstractClass

{

void concreteMethod()

{

// empty

}

abstract void abstractMethod() ;

}

class Access

{

public static void main(String args[ ])

{

new AbstractClass(); **// Error :**

}

}

**Question:** Can we define main() method inside abstract class ?

**Answer:** As main() method is static, JVM no need to instantiate abstract class to invoke main method.

abstract class AbstractClass

{

static void method()

{

System.out.println("static method....");

}

public static void main(String[] args)

{

System.out.println("abstract class main....");

AbstractClass.method();

}

}

* We can access static members of abstract class using class name.
* Hence we can define main method (Static) inside the abstract class.

**Question:** How can we access non-static members of abstract class ?

* Every abstract class must be extended.
* Abstract class specifications (abstract methods) must be implemented in extended class.
* By creating object for child class, we can access the functionality of Parent class(abstract)

abstract class Parent

{

void f1()

{

System.out.println("Concrete method....");

}

abstract void f2();

}

class Child extends Parent

{

void f2()

{

System.out.println("abstract method implementation...");

}

}

class Main

{

public static void main(String[] args)

{

// Parent obj = new Parent(); // Error : we can't instantiate abstract class....

Child obj = new Child();

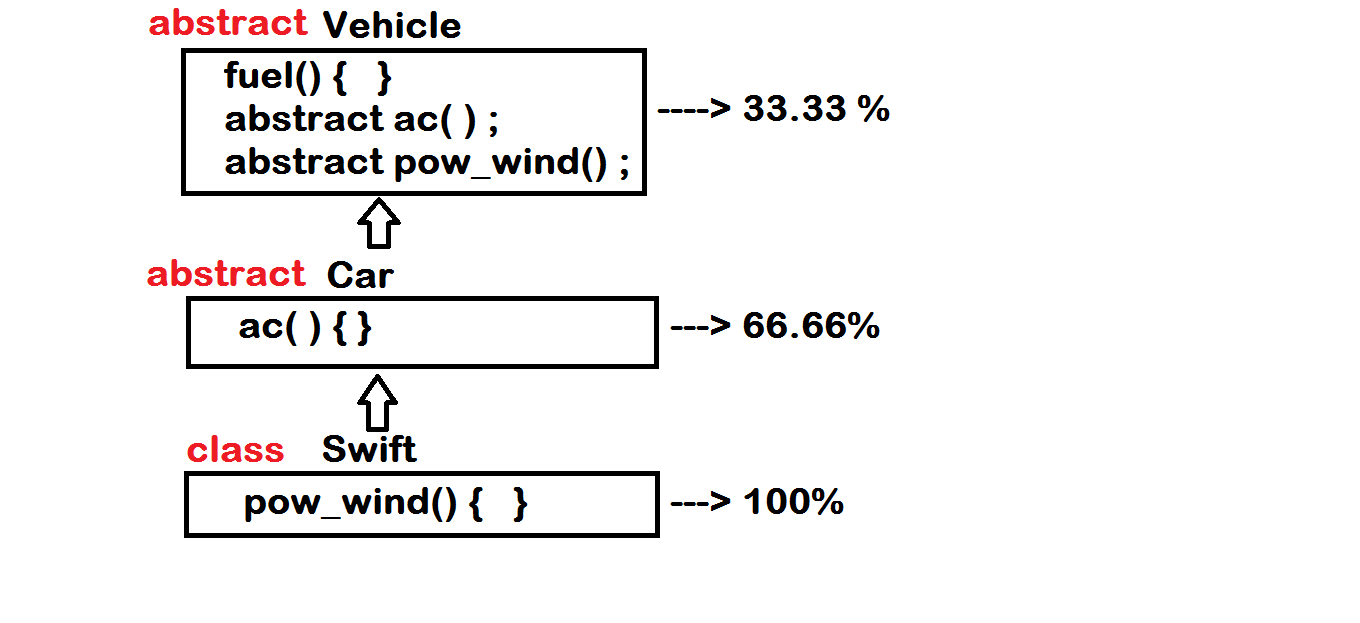
obj.f1();

obj.f2();

}

}

**Note:** When any Child class of abstract class is unable to implement all the specifications of abstract class become another abstract class.



abstract class Vehicle

{

void fuel()

{

System.out.println("fuel....");

}

abstract void ac();

abstract void pow\_wind();

}

abstract class Car extends Vehicle

{

void ac()

{

System.out.println("ac.....");

}

}

class Swift extends Car

{

void pow\_wind()

{

System.out.println("pow\_windows....");

}

}

class Main

{

public static void main(String[] args)

{

Swift car = new Swift();

car.fuel();

car.ac();

car.pow\_wind();

}

}

**Question :** Can we define constructor inside the abstract class ?

**Answer :** Yes allowed, when we instantiate child class, JVM implicitly creates abstract class object by invoking constructor using “super()” method….

abstract class Parent

{

Parent() //concrete method

{

System.out.println("Abstract class instantiation");

}

}

class Child extends Parent

{

Child()

{

// super(); // implicit code...

System.out.println("Child class instantiation");

}

}

class Main

{

public static void main(String[] args)

{

new Child();

}

}

Can we place non-static variables inside abstract class ?

* Yes allowed.
* Non-static variables belongs to Object
* We can’t create object for abstract class directly but it is possible using Child class.
* Hence we provide initial values using super() method from Child class.

abstract class Parent

{

int a ;

Parent(int x)

{

this.a = x ;

}

abstract void details();

}

class Child extends Parent

{

int a ;

Child(int x, int y)

{

super(x);

this.a = y ;

}

void details()

{

System.out.println("Parent's a : "+super.a);

System.out.println("Child's a : "+this.a);

}

}

class Main

{

public static void main(String[] args)

{

int x, y ;

java.util.Scanner sc = new java.util.Scanner(System.in);

System.out.println("Enter Parent's a : ");

x = sc.nextInt();

System.out.println("Enter Child's a : ");

y = sc.nextInt();

Child obj = new Child(x, y);

obj.details();

}

}

**Question:** Can an abstract class be final?

**Answer:** No,

Abstract class must be extended where as

Final class cannot be extended.

final abstract AbstractClass

{

abstract static void method() ;

}

Is-A relation:

final abstract class Parent

{

// logic…

}

class Child extends Parent

{

// logic…

}

Note : A method cannot be abstract & final

A method cannot be abstract & static

class Test

{

abstract final void f1(); // Error

abstract static void f2(); // Error

}

**Question:** Why abstract method cannot be final?

**Answer :** Abstract method must be overridden in its sub class, but if method is final, we cannot override. Hence it is illegal combination.

**Question:** Why abstract method cannot be static?

* Common functionality of Object we define as static.
* Abstract methods are specific to particular object.
* Hence both static (common) and abstract (specific) are illegal combination.

**Question:** Illegal combination of modifiers?

1. abstract & final
2. abstract & static

**Question:** Can a final class extends Abstract class?

**Answer:** Yes,

but it has to implement all the specifications of abstract class

because another extension of final class is not allowed.

abstract class AbstractClass

{

abstract void f1();

abstract void f2();

}

final class FinalClass extends AbstractClass

{

void f1(){ }

void f2(){ }

}

**Interfaces**

**class:** Complete representation of Object(100%)

(Only concrete methods)

**abstract class:** Partial representation of Object(.....%)

(Both Concrete & Abstract methods)

**interface:** Complete specification of Object (0%)

(Only abstract methods)

* “interface” is a pre-defined modifier is used to define set of specifications.
* “interface” definition allow only abstract methods.

interface Test

{

void method()

{

// Error : interface methods can't have body

}

}

**By default interface methods are “public & abstract”**

interface Test

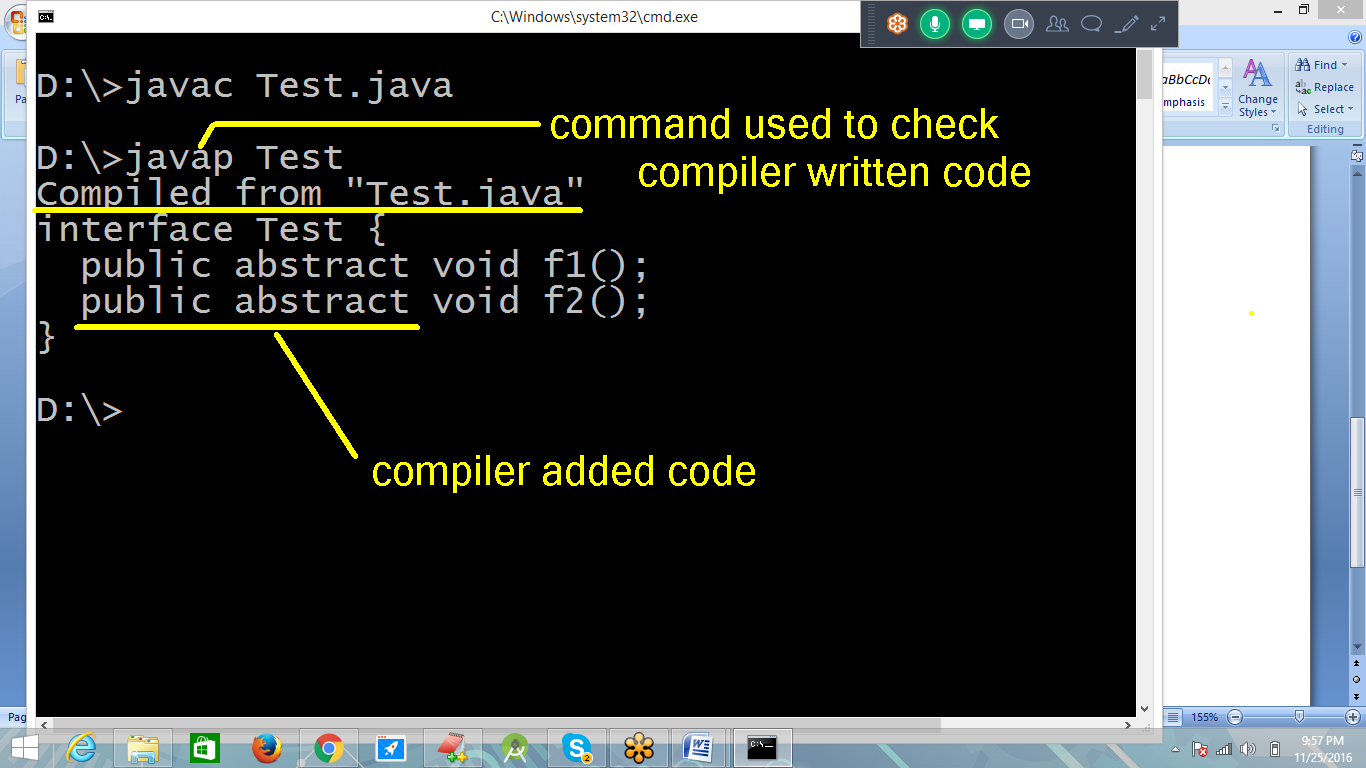
{

void f1() ; // by default public & abstract

void f2() ;

}

* To check compiler added code to the class file we use javap command.



* We can’t instantiate interface.
* Interface is not fully defined.
* Interface doesn’t allowed constructors.
* We can’t instantiate without constructor.

interface Test

{

void f1() ;

void f2() ;

}

class Main

{

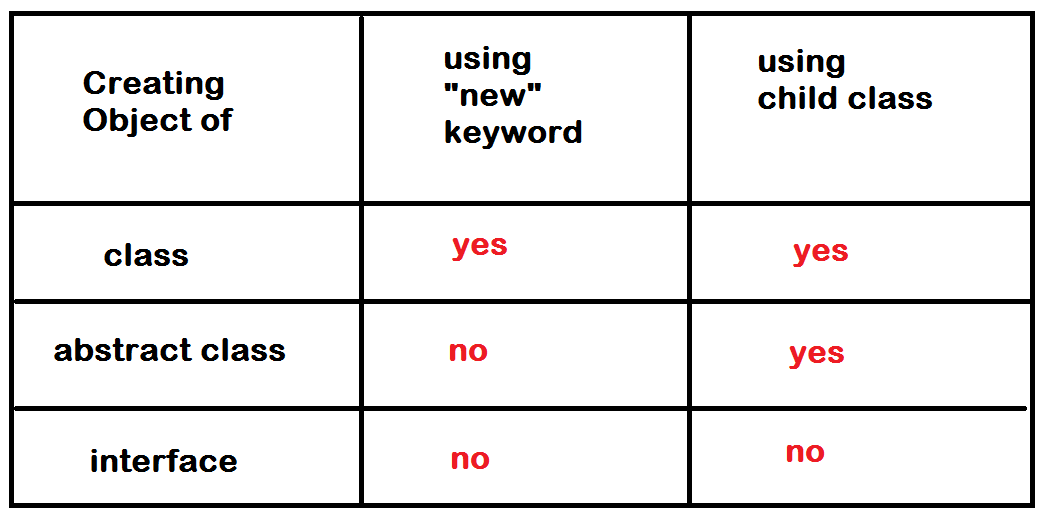
public static void main(String args[])

{

new Test();// Error : can't instantiate

}

}



Question : Can we define main method inside interface ?

Ans : It is allowed since jdk 1.8

interface Test

{

public static void main(String args[ ])

{

System.out.println("interface main method.....");

}

}

* We can define static methods and can access directly using interface identity.
* Abstraction is applicable for non static members(methods).
* Static members cannot be overridden because it is common functionality of Object.
* We cannot update static members in the hierarchy.
* If we want to update, we should modify in the top class of hierarchy.

interface Test

{

static void m()

{

System.out.println("interface static method");

}

public static void main(String args[])

{

System.out.println("interface main method....");

Test.m();

}

}

* Only static method definitions allowed inside interface.
* Static methods cannot be overridden.

Note : Any class can “**implements**” interface but not “**extends**”.

* Static methods are not specific.

interface Test

{

void f1() ; // public method

void f2() ;

}

class Main implements Test

{

// void f1(){ } // Error : pls don't decrease privileges

public void f1(){ } // must override as public

public void f2(){ }

public static void main(String args[])

{

Main obj = new Main();

obj.f1();

obj.f2();

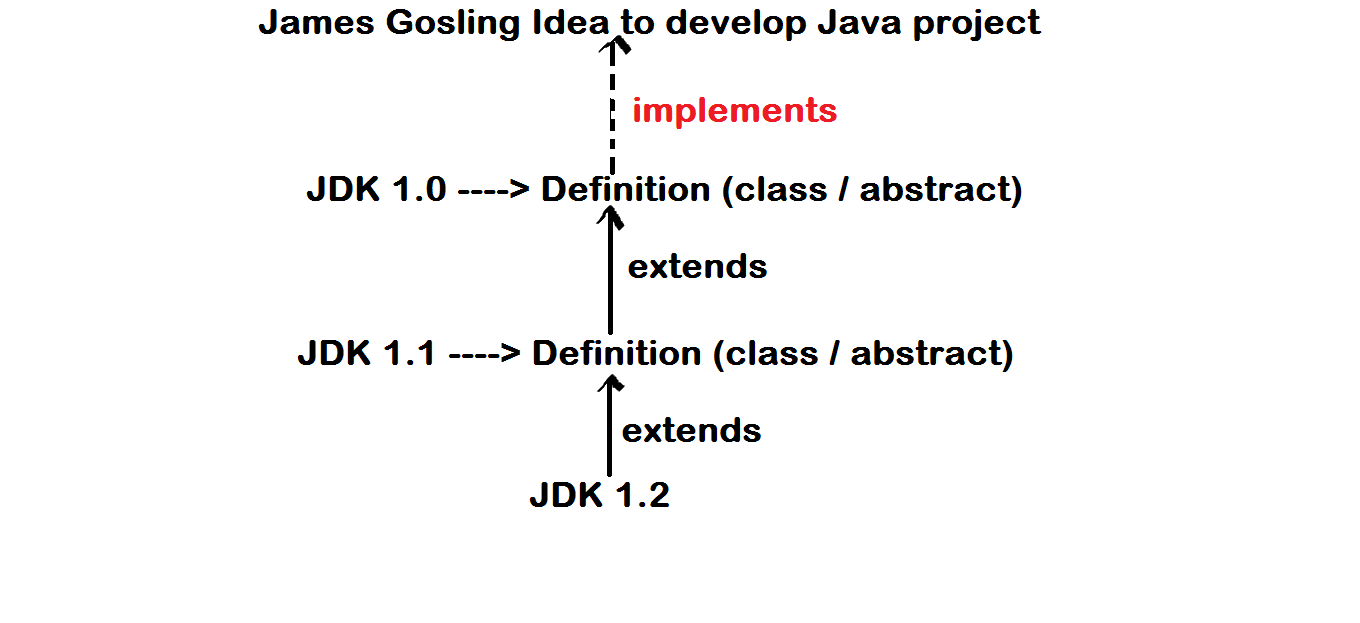
}

}

**Ques : Why a class “implements” interface instead of “extends” ?**

Ans : In case of class or abstract class, we can extend(update) existing functionality.

But in case of “interface” only specifications are available to implement.



Note: If any class is unable to “implement” all the specifications of interface must be defined as “abstract class”.

interface In // 0%

{

void m1();

void m2();

}

abstract class One implements In // 50%

{

public void m1(){ }

}

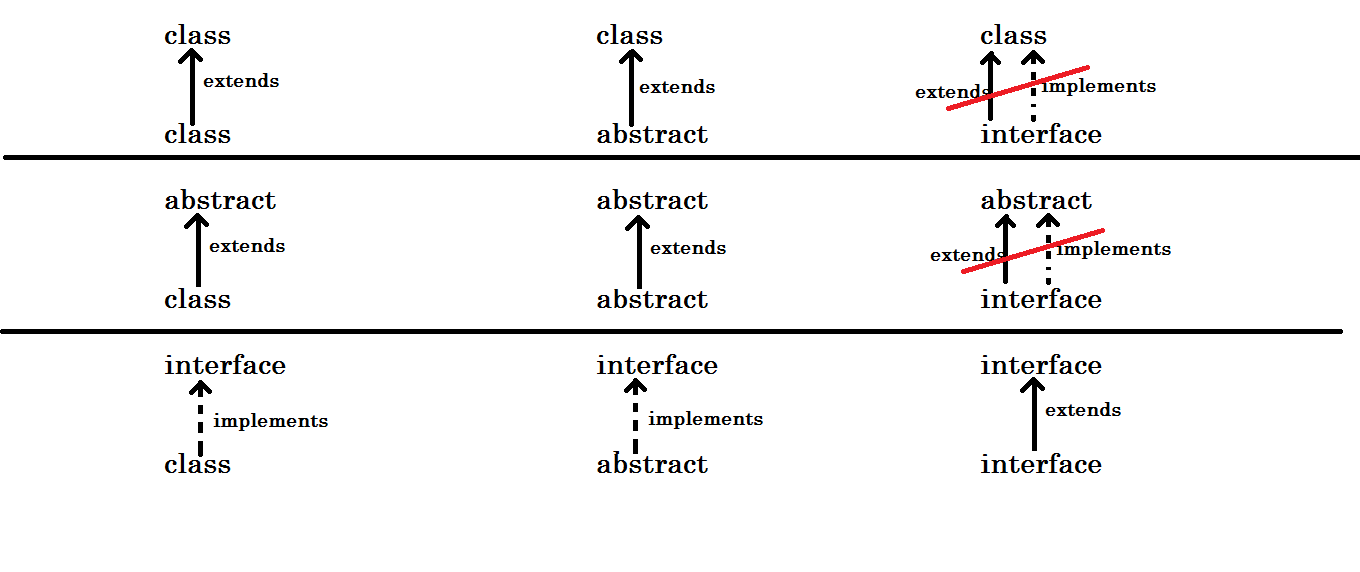
class Two extends One // 100%

{

public void m2(){ }

}

**Relations among class, abstract class & interface :**



**Note:** We can implement “Multiple Inheritance” using interfaces in Java application.

One class can extends only one class.

One class can implement more than one interface.

**One interface can extends many interfaces (Multiple inheritance).**

Following diagram describes the relation between classes, abstract class and interfaces.

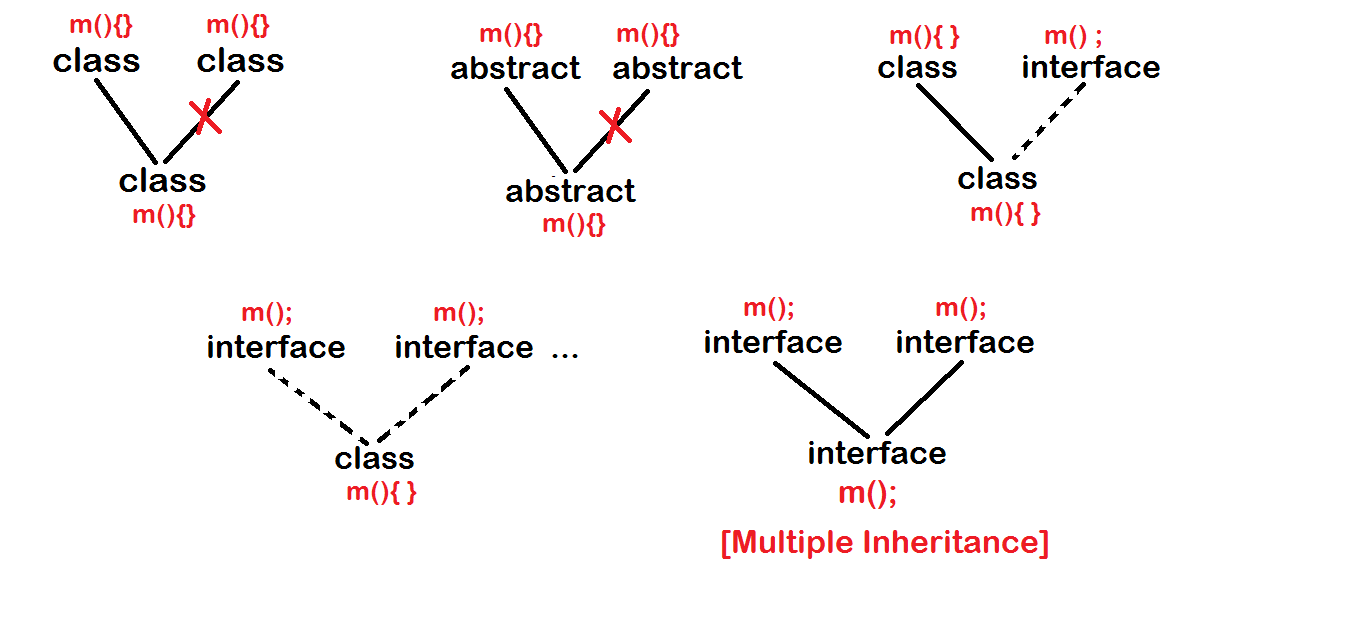


Diagram1:

class A

{

}

class B

{

}

class Diagram1 extends A, B

{

}

Diagram2:

abstract class A

{

}

abstract class B

{

}

abstract class Diagram2 extends A,B

{

}

Diagram3:

class A

{

}

interface B

{

}

class Diagram3 extends A implements B

{

}

Diagram4:

interface A

{

}

interface B

{

}

class Diagram4 implements A, B

{

}

Diagram5 – Multiple inheritance:

interface A

{

}

interface B

{

}

interface Diagram5 extends A, B

{

}

Upcasting:

* The concept of storing Child class object address into Parent type reference variable.
* Using Parent address, we can access common functionality or overridden functionality of Both Parent and Child.

class Parent

{

void common()

{

System.out.println("Parent's common");

}

}

class Child extends Parent

{

void common()

{

System.out.println("Child's common");

}

}

class Upcast

{

public static void main(String[] args)

{

Parent addr = new Child(); // upcasting...

addr.common();

}

}

Using Parent, we cannot access Child’s specific functionality (Down casting is required).

class Parent

{

void common()

{

System.out.println("Parent's common");

}

}

class Child extends Parent

{

void common()

{

System.out.println("Child's common");

}

void specific()

{

System.out.println("Child's specific");

}

}

class Upcast

{

public static void main(String[] args)

{

Parent addr = new Child();

addr.common();

addr.specific(); // Error :

}

}

What is the use of upcasting?

* In all the situations, we cannot access object functionality directly.
* We use Parent type reference variable to identify the Children.

Parent address = new Child();

Hostel address = new Student();

Owner address = new Tenant();

Institute address = new Faculty();

Interfaces in upcasting:

* Interface is a set of specifications.
* We implement any interface with a class.
* Once we have implemented, we need to label the object with interface name.
* Interface is a standard.
* In communication, we use standard objects for communication.

interface Lenovo

{

void processor();

void motherBoard();

void display();

}

class Computer implements Lenovo

{

public void processor(){ }

public void motherBoard(){ }

public void display(){ }

}

class Upcast

{

public static void main(String[] args)

{

Lenovo obj = new Computer(); // upcast

obj.display();

obj.processor();

obj.motherBoard();

}

}

* When we implement an object from different set of standards (interfaces), we cannot give a particular name to access the complete functionality.
* We label the object with implemented class name only.

interface Intel

{

void processor();

}

interface Asus

{

void motherBoard();

}

interface Samsung

{

void display();

}

class AssembledComputer implements Intel, Asus, Samsung

{

public void processor(){ }

public void motherBoard(){ }

public void display(){ }

}

class Upcast

{

public static void main(String[] args)

{

AssembledComputer obj = new AssembledComputer();

obj.display();

obj.processor();

obj.motherBoard();

Samsung comp = new AssembledComputer();

comp.display();

comp.processor(); // Error :

}

}

**Up casting:**

Parent obj = new Child();

**Down casting:**

* It is the concept of collecting Child object address back to Child type reference variable from Parent type variable.

Child obj = new Parent(); // It is not….

Parent obj = new Child(); // upcast

Child obj1 = (Child)obj ; // downcast

* Downcast is explicit(manual)
* Using downcast, we can access specific functionality of Child object.

class Parent

{

void common()

{

System.out.println("Parent's common");

}

}

class Child extends Parent

{

void specific()

{

System.out.println("Child's specific");

}

}

class Downcast

{

public static void main(String[] args)

{

Parent p = new Child();

p.common();

// p.specific(); -> Error :

Child c = (Child)p; // downcast

c.specific();

}

}

* We can implement down casting with overriding concept only.
* Defining abstract method is also the concept of overriding.

interface Parent

{

void childMobileNum();

}

abstract class Child implements Parent

{

public abstract void friendMobileNum();

public void childMobileNum()

{

System.out.println("Child Mobile Num : 9988776655");

}

}

class ChildFriend extends Child

{

public void friendMobileNum()

{

System.out.println("Friend Mobile Num : 7755884477");

}

}

class Downcast

{

public static void main(String[] args)

{

Parent p = new ChildFriend();

p.childMobileNum();

// p.friendMobileNum(); -> Error :

Child c = (Child)p ;

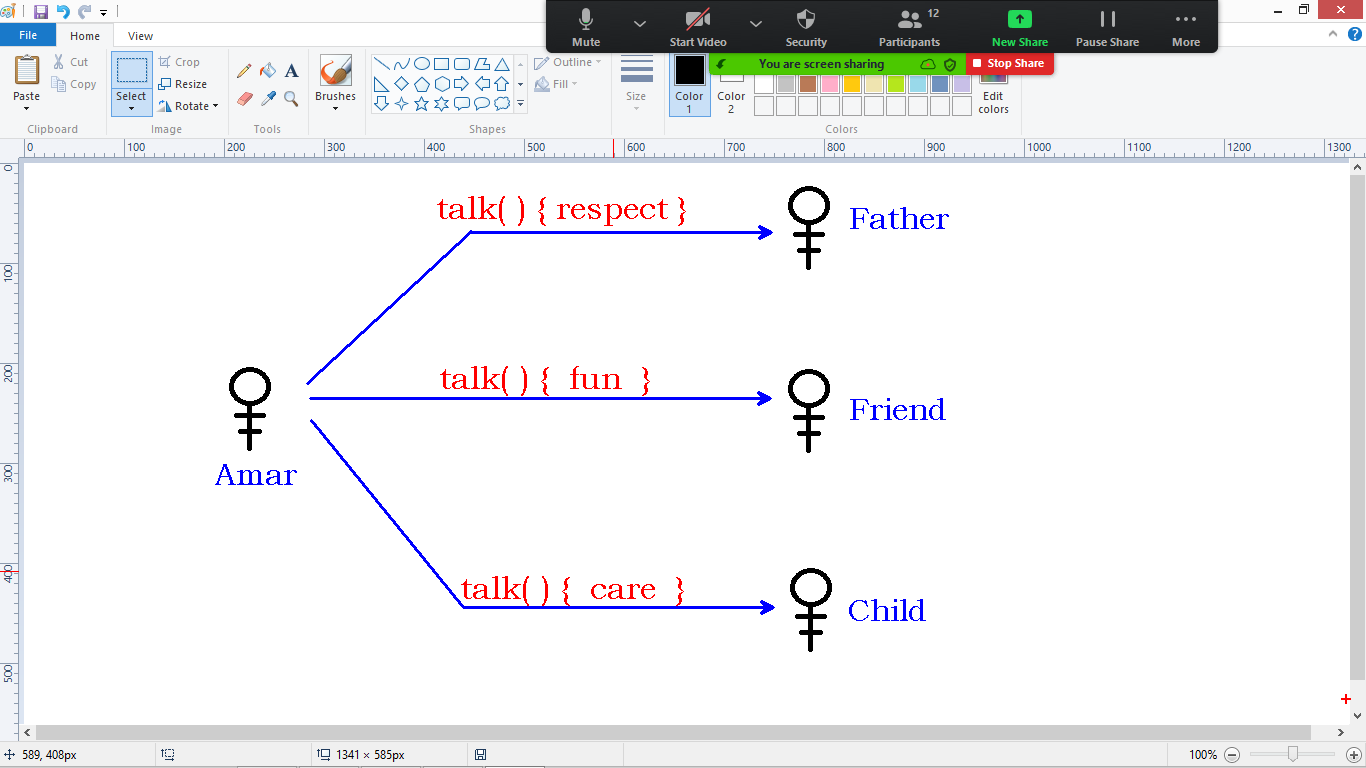
c.friendMobileNum();

}

}

**Polymorphism**

* Defining an object (class) that shows different behavior (methods) with same identity.
* ‘Poly’ means ‘Many’
* ‘Morphic’ means ‘Forms’



* Polymorphism is of 2 types
  + Compile time Polymorphism
  + Runtime polymorphism

1. Compile time:
   1. It is also called static binding.
   2. It is method overloading technique.
   3. Overloading – Defining more than one method with the same name but with different signature(arguments)
   4. ‘static’ means ‘input type based’
   5. ‘binding’ means ‘connecting’
   6. Overloading belongs to same class.

**Note:** Compiler can bind up with the method at the time of compilation based on the input argument values in method call.

class Calculator

{

void add(int x, int y)

{

System.out.println("Sum of 2 num's : " + (x+y));

}

void add(int x, int y, int z)

{

System.out.println("Sum of 3 num's : " + (x+y+z));

}

void add(String s1, String s2)

{

System.out.println("After concat : " + (s1+s2));

}

}

class OverLoading

{

public static void main(String[] args)

{

Calculator calc = new Calculator();

calc.add(10,20);

calc.add(10,20,30);

calc.add("Online" , "Class");

}

}

Can we overload constructor?

* Yes allowed.
* POJO class
* Constructor chaining.

class Test

{

Test()

{

System.out.println("Zero args");

}

Test(int x)

{

this();

System.out.println("Args");

}

public static void main(String[] args)

{

new Test(10);

}

}

Can we overload main() method?

* Yes allowed.
* We can overload main() but we need to access explicitly.
* JVM implicitly access standard main() method only.

class Overload

{

void main()

{

System.out.println("Zero args");

}

static void main(int x)

{

System.out.println("Args");

}

public static void main(String args[ ])

{

System.out.println("JVM main");

Overload obj = new Overload();

obj.main();

obj.main(10);

}

}

Can we call standard main() explicitly?

* Yes allowed.

class Overload

{

static

{

System.out.println("Block starts...");

String arr[] = {"one" , "two" , "three"};

Overload.main(arr);

System.out.println("Block ends...");

}

public static void main(String args[ ])

{

System.out.println("JVM main");

}

}

1. Runtime polymorphism:

* It is also called dynamic binding.
* It is method overriding technique.
* Overriding : Defining a method in Child class with the same name and same signature of Parent class.
* It must be implemented only in Parent-Child relation.
* Using “Child” object we can access the functionality of all classes in the hierarchy.

class Parent

{

void fun()

{

System.out.println("Parent's fun");

}

}

class Child extends Parent

{

void fun()

{

System.out.println("Child's fun");

}

void behavior()

{

super.fun();

this.fun();

}

}

class Overriding

{

public static void main(String[] args)

{

Child obj = new Child();

obj.behavior();

}

}

Why it is called Runtime binding?

* It is possible to store Child object address into Parent type variable – Up casting.
* In up casting, compiler directly connects to Parent object but the Child functionality executes at runtime if available.
* Hence compiler cannot connect with the functionality directly.

class Parent

{

void fun()

{

System.out.println("Parent's fun");

}

}

class Child extends Parent

{

void fun()

{

System.out.println("Child's fun");

}

}

class Overriding

{

public static void main(String[] args)

{

Parent obj = new Child();

obj.fun();

}

}

**Polymorphism**

Defining an Object(class) that shows different behavior(methods) with the same Identity.

“Poly” means "Many"

“Morphic” means "Forms"



Java supports 2 types of polymorphism:

1. Compile time polymorphism
2. Runtime polymorphism

**Compile time polymorphism:**

* It is also called static binding.
* It is method overloading technique.
* Method overloading refers defining more than one method with the same name within the same class but with different signatures.
* It can be implemented in a single class.
* Hence single object can alone shows compile time polymorphism behavior.
* Compiler connects with the method dependes on method call. So at runtime, JVM no need to search for the method to be executed.

class Addition

{

void sum(int x, int y)

{

System.out.println("Sum of 2 int's : "+(x+y));

}

void sum(int x, int y, int z)

{

System.out.println("Sum of 3 int's : "+(x+y+z));

}

}

class StaticBinding

{

public static void main(String[] args)

{

Addition obj = new Addition();

obj.sum(10,20);

obj.sum(10,20,30); // Method call

}

}

Can we overload Constructor?

* Yes allowed.
* Constructor is a special method.
* We can define more than one constructor with different arguments.
* We can connect them using this() method is called Constructor chaining.

class Test

{

Test()

{

this(10); // invokes args constructor

System.out.println("No args constructor...");

}

Test(int x)

{

System.out.println("Args constructor...");

}

public static void main(String[] args)

{

new Test(); // invokes zero args

}

}

Can we overload main() method ?

* Yes allowed.
* JVM invokes only standard main method implicitly
* User main methods must be accessed manually by the programmer.

class OverLoad

{

void main()

{

System.out.println("Zero args main....");

}

/\*static void main()

{

Error : Not allowed, signature should be different.

}\*/

static void main(int x)

{

System.out.println("One arg main...");

}

public static void main(String args[ ])

{

System.out.println("Standard JVM method....");

OverLoad obj = new OverLoad();

obj.main();

obj.main(10); // OverLoad.main();

}

}

**Runtime polymorphism:**

* It is called Dynamic binding.
* It is Method overriding technique.Defining a method in Child class with the same name and same signature of its Parent class is called Method overriding.
* We can implement dynamic binding only in “Is-A” relation.
* More than one object is combinely showing Runtime polymorphism behavior.

**Note:** One advantage of Runtime Polymorphism is used to “Update Parent class functionality in Child class” if it is not sufficient.

class Galaxy

{

void camera(){

12 mega pixel ;

}

}

class Note extends Galaxy

{

void camera(){

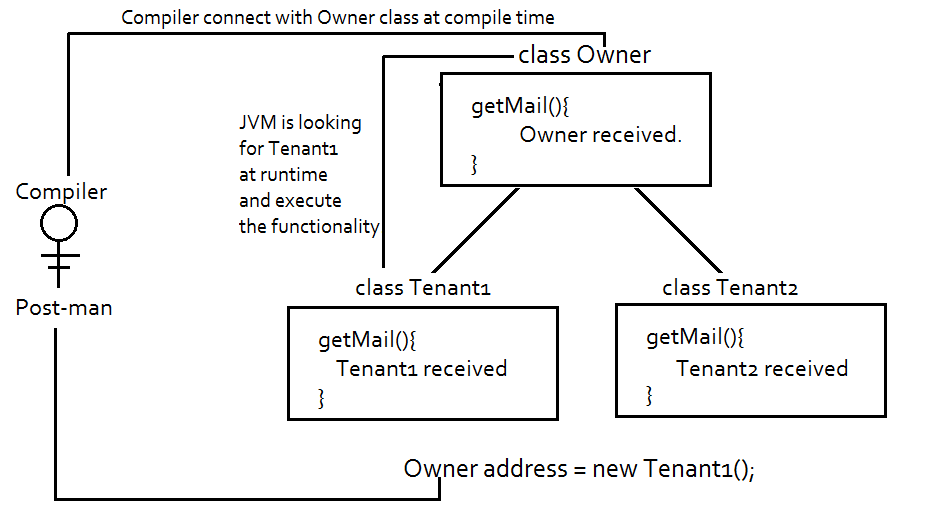
20 mega pixel ; // updating the functionality

}

}

The main advantage of Runtime polymorphism is “Accessing Child class functionality using Parent reference”.

* + Parent ref = new Child();
  + Hostel addr = new Student();
  + Owner addr = new Tenant();
  + College addr = new Faculty()
* We know the importance of “C/O address” while addressing someone.
* The best example of runtime polymorphism as follows.



class Owner

{

void sendMail()

{

System.out.println("Owner received the mail....");

}

}

class Tenant1 extends Owner

{

void sendMail()

{

System.out.println("Tenant1 received the mail....");

}

}

class Tenant2 extends Owner

{

/\* void sendMail()

{

System.out.println("Tenant2 received the mail....");

} \*/

}

class DynamicBinding

{

public static void main(String[] args)

{

Owner address = new Tenant1();

address.sendMail();

Owner address2 = new Tenant2();

address2.sendMail();

}

}

Why it is allowed to access sub class functionality using its super class?

* In Java hierarchy(single, multi-level or hierarachal), from any sub class to super class, only one path is available to upcast. Hence no chance of deviation.

**Question:** why object up casting is implicit?

* Because multiple inheritance is not supported in java.
* No chance of deviation while up casting of object as there is only one parent for multiple child classes in the hierarchy.
* But down casting must be done explicitly.

class Parent

{

}

class Child1 extends Parent

{

}

class Child2 extends Parent

{

}

class DownCasting

{

public static void main(String[] args)

{

Parent p1 = new Child1();

Parent p2 = new Child2();

// Child1 c1 = p1 ; //Error :

// Child2 c2 = p2 ; //Error :

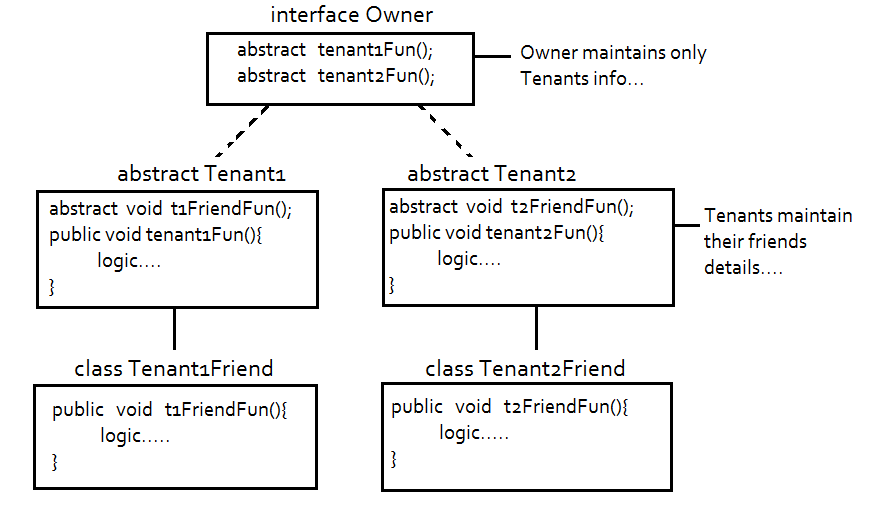
Child1 c1 = (Child1)p1 ; //Down casting...

Child2 c2 = (Child2)p2 ;

}

}

* We can implement Runtime Polymorphism using Interfaces & Abstract classes.
* To connect runtime polymorphism Object, we can use simply specifications(abstract methods)
* As shown in the diagram,
  + We can access Tenant1 using Owner
  + We can’t access Tenant1Friend functionality using Owner
  + Suppose if we pass a mail to Tenant1Friend using Owner, results Error.
  + If “Owner” pass that information to Tenant1 (Downcast), then accessing is possible.
  + Please go through diagram and implementation carefully.



interface Owner

{

void tenantFun();

}

abstract class Tenant implements Owner

{

public abstract void tenantFriendFun();

public void tenantFun()

{

System.out.println("Accessing Tenant Functionality....");

}

}

class TenantFriend extends Tenant

{

public void tenantFriendFun()

{

System.out.println("Accessing tenant friend functionality....");

}

}

class DownCasting

{

public static void main(String[] args)

{

Owner owner = new TenantFriend();

owner.tenantFun(); /\* We can access tenant using owner \*/

// owner.tenantFriendFun(); /\* Error : We cannot access \*/

Tenant tenant = (Tenant)owner ; /\* Owner passing information to Tenant \*/

tenant.tenantFriendFun();

}

}

**Exception Handling**

During program Compilation and Execution, chance of getting 3 kinds of Errors

1. Compile time errors
2. Logical errors
3. Runtime errors – Exceptions

**1) Compile time errors:**

* Every programming language follows a set of rules.
* We need to implement those rules while defining source code.
* Compiler raises error if we violate those rules.
* Language rules also called “syntax rules”.

**Examples:**

1. Cannot find symbol
2. “this” cannot be used in static context
3. Method should have return type.
4. Abstract and Final cannot combined
5. Call to super() must be the first statement.

abstract class Test

{

abstract final void fun();

}

**2) Logical errors:**

* No error at the time of compilation and execution.
* The code produce unexpected output, hence the program is logically incorrect.

class Test

{

public static void main(String[] args)

{

int a=5, b=2 ;

float c = a/b ; // Logic error : int/int = int

System.out.println("Result : " + c);

}

}

class Test

{

public static void main(String[] args)

{

int a=5, b=2 ;

float c = (float)a/b ; // float/int = float

System.out.println("Result : " + c);

}

}

**3) Runtime errors:**

* Getting error while application is running.
* Runtime errors also called Exceptions.
* Exception is a “class”.
* Java library given number of exception classes.
* Programmer can define user exceptions if required.



* Programmer need to define code for success and failure cases.
* Failure case always raises exception.
* Runtime error causes:
  + Abnormal termination of program
  + Informal information to End user.

import java.util.Scanner;

class ReadInt

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

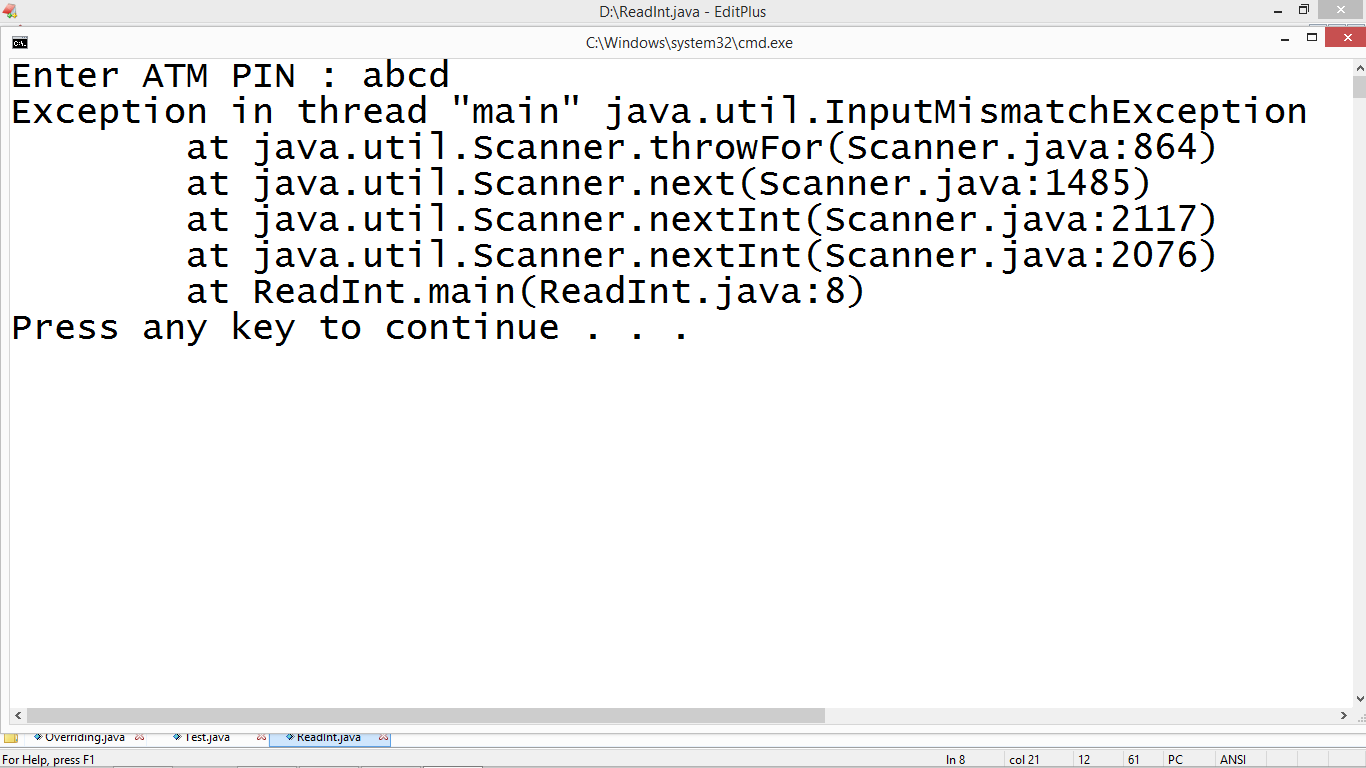
System.out.print("Enter ATM PIN : ");

int pin = scan.nextInt();

System.out.println("PIN is : " + pin);

}

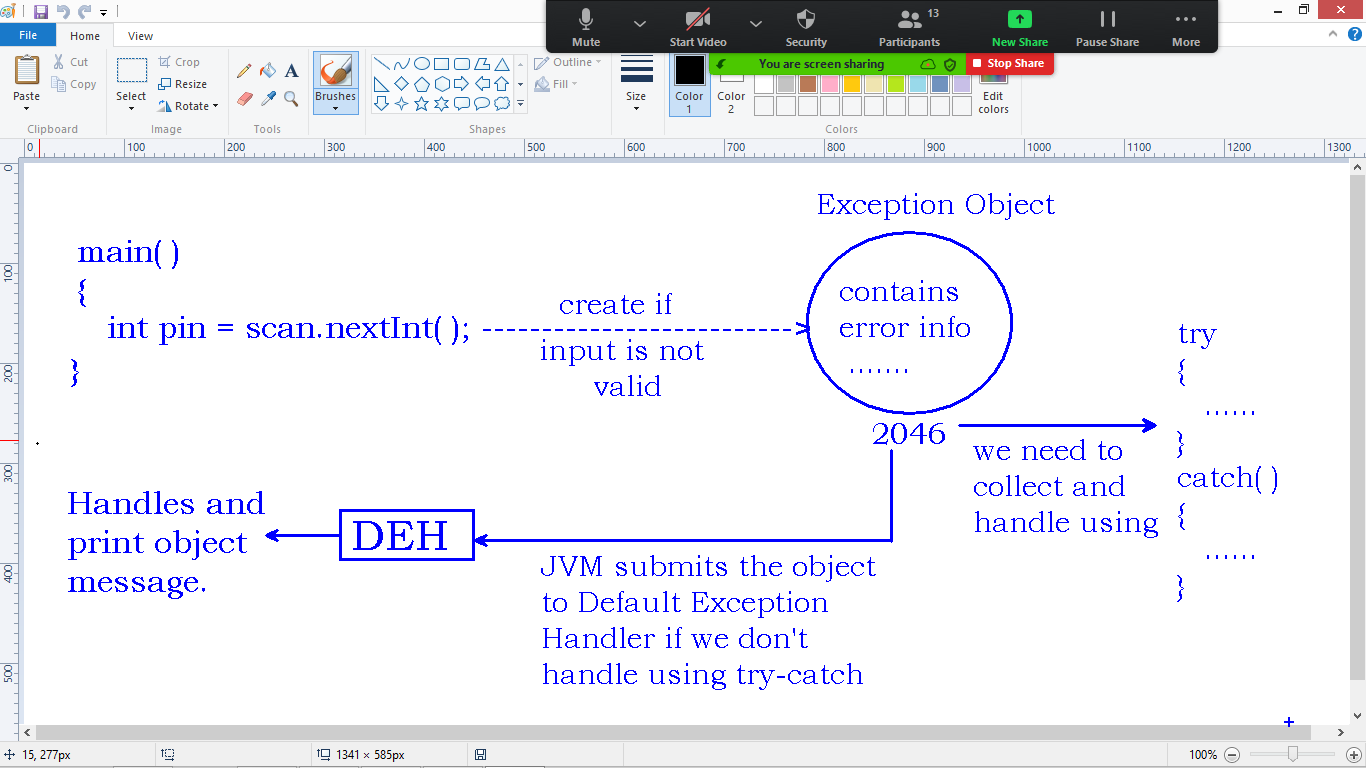
}



* When exception has risen, an object will be created with the complete information of Runtime error.
* We need to collect the object and handle to avoid abnormal termination of program.
* To handle, java library given a set of keywords
  + Try
  + Catch
  + Finally
  + Throw
  + Throws

**Default Exception Handler:**

* It is a pre-defined program
* It is recommended to handle every exception by the programmer.
* If the programmer is not handled then “Default Exception Handler” handles the object with pre-defined error information.



**Handling Exception:**

* Try block contains doubtful code that raises exception.
* When exception raises, try block throws exception object to catch block.
* Catch block collects the exception object and handles.
* If no exception in try block, catch block will not execute.

import java.util.Scanner;

import java.util.InputMismatchException;

class ReadInt

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

try

{

System.out.print("Enter ATM PIN : ");

int pin = scan.nextInt();

System.out.println("PIN is : " + pin);

}

catch (InputMismatchException e)

{

System.out.println("Exception : Invalid input type");

}

}

}

**Try with multi catch:**

* One try block can have multiple catch blocks.
* In try block, we can define number of instructions.
* Try block can raise more than one exception from its logic.
* To handle different exceptions, more than one catch is required.

**Note:** Only one catch block executes among we defined.

import java.util.Scanner;

import java.util.InputMismatchException;

class Division

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

try

{

System.out.println("Enter 2 numbers : ");

int x = scan.nextInt();

int y = scan.nextInt();

int z = x/y;

System.out.println("Result : " + z);

}

catch (InputMismatchException e1)

{

System.out.println("Exception : Invalid input type");

}

catch (ArithmeticException e2)

{

System.out.println("Exception : Invalid division");

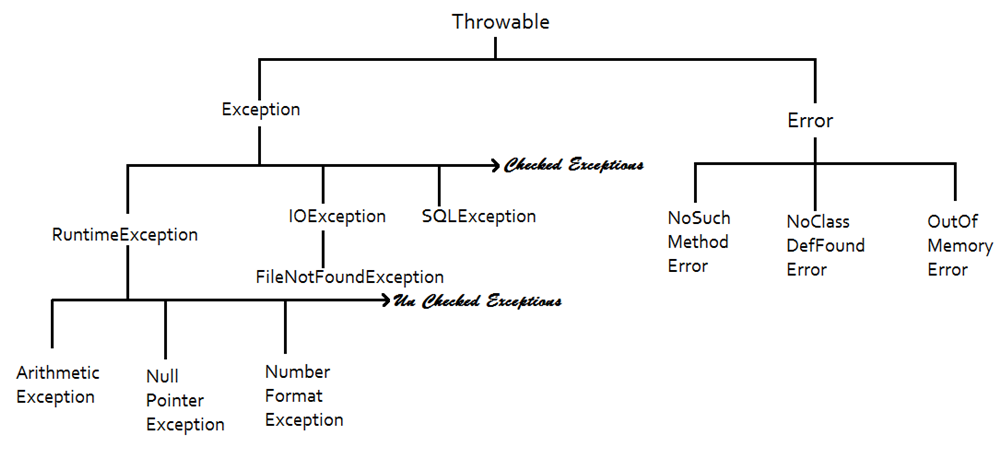
}

}

}

Exception Hierarchy:

* ‘Throwable’ is a super classes of all Exception classes.
* Throwable has 2 children called Exception & Error.
* We cannot handle Errors.
* We can handle only Exceptions.
* Exception classified into
  + Checked Exceptions
  + Unchecked Exceptions
* Child class of RuntimeException becomes unchecked exception.



**Why we cannot handle errors?**

* Exception is a runtime error.
* Exception occurs while application is running.
* In case of Error, we cannot run the application.
* For example.
  + If main() is not defined, we cannot execute the code called “Error”
  + If the memory is Full, program cannot run called “Error”
  + If the class is not present that we are accessing results “Error”

class Test

{

// no main...

}

Compile: No error

Run: **Error** - no main() method

* We can handle the exception by collecting the exception object into its Parent type variable also.
* This concept is called Object up casting.

import java.util.\*;

class ReadInt

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

try

{

System.out.print("Enter ATM PIN : ");

int pin = scan.nextInt();

System.out.println("PIN is : " + pin);

}

// catch (InputMismatchException e)

// catch (RuntimeException e)

// catch (Exception e)

catch (Throwable e)

{

System.out.println("Exception : Invalid input type");

}

}

}

* We can handle more than one exception using single catch block.
* Multiple exceptions can be handled by specifying Parent Exception type in the catch block.

import java.util.Scanner;

import java.util.InputMismatchException;

class Division

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

try

{

System.out.println("Enter 2 numbers : ");

int x = scan.nextInt();

int y = scan.nextInt();

int z = x/y;

System.out.println("Result : " + z);

}

catch (Exception e)

{

System.out.println("Exception Raised - Handled");

}

}

}

getMessage():

* A pre-defined method belongs to every exception class.
* When exception has risen, an object will be created with error information.
* When we invoke getMessage(), the object error information will be returned.

import java.util.Scanner;

import java.util.InputMismatchException;

class Division

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

try

{

System.out.println("Enter 2 numbers : ");

int x = scan.nextInt();

int y = scan.nextInt();

int z = x/y;

System.out.println("Result : " + z);

}

catch (Exception e)

{

System.out.println("Exception : " + e.getMessage());

}

}

}

**Java Error classes:**

* An Error is a subclass of Throwable.
* An Error class indicates serious problems that a reasonable application should not try to catch.
* Most of the errors are abnormal conditions.

The java.lang.Errors provides for different errors thrown under java lang package.

**NoClassDefFoundError :** This is Thrown if the Java Virtual Machine or a ClassLoader instance tries to load in the definition of a class and no definition of the class could be found.

**NoSuchMethodError :** This is Thrown if an application tries to call a specified method of a class (either static or instance), and that class no longer has a definition of that method.

**OutOfMemoryError :** This is Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

**Why we cannot handle errors?**

* Exception is a runtime error.
* Exception occurs while application is running.
* In case of Error, we cannot run the application.
* For example.
  + If main() is not defined, we cannot execute the code called “Error”
  + If the memory is Full, program cannot run called “Error”
  + If the class is not present that we are accessing results “Error”

class Test

{

// no main...

}

Compile: No error

Run: **Error** - no main() method

**Unchecked v/s Checked exceptions:**

* Exceptions classified into Checked and Unchecked.
* Unchecked exceptions handling is optional.
* Checked exceptions must be handled else compiler raises error.

**Unchecked Exceptions:**

* The child class of RuntimeException is called Unchecked.
* Examples…
  + ArithmeticException
  + NumberFormatException
  + NullPointerException
  + InputMismatchException
  + ….
* In case of Unchecked, java program is not connected to any resource

**Checked Exceptions:**

* Class which is not the Child class of RuntimeException.
* Examples…
  + FileNotFoundException : java code connected to File
  + IOException : java code connected to IO device
  + SQLException : Java code connected to Database
  + ServletException : Java code connected to Server application.
* If we don’t handle the checked exception, compiler will not generate class file, hence we cannot run program.
* After working with any resource in the application, we need to release that code properly.
* Improper shutdown of resources causes loss of data.

**Checked Exception code:**

* Opening a file in Read mode.
* Java.io package is providing FileInputStream class to open a file in read mode.

class FileInputStream

{

public FileInputStream([String](file:///G:\pendrive\My%20Web%20Sites\java%20api\docs.oracle.com\javase\7\docs\api\java\lang\String.html) name) throws [FileNotFoundException](file:///G:\pendrive\My%20Web%20Sites\java%20api\docs.oracle.com\javase\7\docs\api\java\io\FileNotFoundException.html)

{

Input argument is “File path”

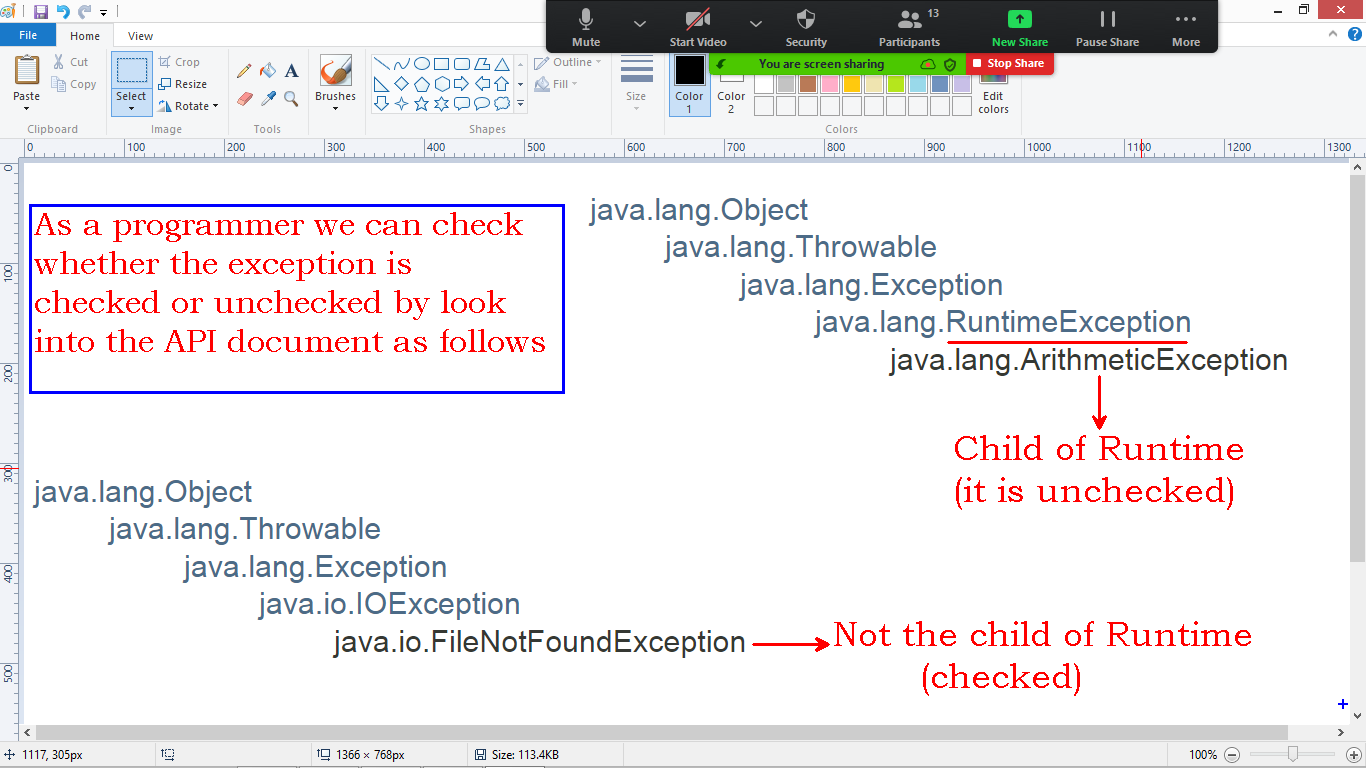
If the file is present, it will open the file in read mode.

Raises exception if the file is not present.

“FileNotFoundException” is checked hence must be handled.

}

}



* We will get Compile time error in the below code as we are not handling exception.

import java.io.FileInputStream;

class OpenFile

{

public static void main(String[] args)

{

String path = "e:/test.txt";

FileInputStream file = new FileInputStream(path);

}

}

* We need to handle FileNotFoundException to avoid compile time error.

import java.io.FileInputStream;

import java.io.FileNotFoundException;

class OpenFile

{

public static void main(String[] args)

{

String path = "e:/test.txt";

try

{

FileInputStream file = new FileInputStream(path);

System.out.println("File opened...");

}

catch (FileNotFoundException e)

{

System.out.println("Exception : " + e.getMessage());

}

}

}

**How to close the file?**

**Answer :** Finally block is used to release the resource which is connected to program code.

**Finally:**

* ‘finally’ is a keyword.
* ‘finally’ is a block of statements.
* ‘finally’ block is used to release(close) any resource connected to program.
* Finally block executes whether or not an exception has raised in try block.

class FinallyBlock

{

public static void main(String[] args)

{

try

{

int x = 10/5 ;

System.out.println("try");

}

catch (Exception e)

{

System.out.println("catch");

}

finally

{

System.out.println("finally");

}

}

}

class FinallyBlock

{

public static void main(String[] args)

{

try

{

int x = 10/0 ;

System.out.println("try");

}

catch (Exception e)

{

System.out.println("catch");

}

finally

{

System.out.println("finally");

}

}

}

* We can define try & finally blocks without catch block.
* Catch block is used to collect the exception object to be handled.
* If we don’t catch, the object will be handled by Default exception handler.

class FinallyBlock

{

public static void main(String[] args)

{

try

{

int x = 10/5 ;

System.out.println("try");

}

finally

{

System.out.println("finally");

}

}

}

* If we don’t define the catch block and exception has risen, the program terminates abnormally.
* ‘finally’ block executes in case of abnormal termination also….

class FinallyBlock

{

public static void main(String[] args)

{

try

{

int x = 10/0 ;

System.out.println("try");

}

finally

{

System.out.println("finally");

}

}

}

**Open and Close a File:**

* java.io.FileInputSteram class is providing pre-defined function close().
* close() method is used to close the file which associated with the current stream.

/\*

class FileInputStream

{

public void close() throws IOException

{

Close the file.

}

}

\*/

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

class OpenFile

{

static FileInputStream file = null ;

public static void main(String[] args)

{

String path = "e:/test.txt";

try

{

file = new FileInputStream(path);

System.out.println("File opened...");

}

catch (FileNotFoundException e)

{

System.out.println("Exception : " + e.getMessage());

}

finally

{

try

{

file.close();

System.out.println("File closed...");

}

catch (IOException io)

{

System.out.println("IO error");

}

}

}

}

**Custom Exceptions:**

* Java API providing pre-defined exception classes.
* Every programmer can define user exceptions also.
* A custom exception extends the functionality from pre-defined exception classes.
* Custom exception can be either Checked or Unchecked.
* Most of the custom exceptions are Checked.

class CustomException extends RuntimeException

{

// user defined unchecked exception

}

class CustomException extends Exception

{

// user defined checked exception

}

**How we construct exception object with Error-information?**

* Every exception class has pre-defined constructor with String argument.
* We need to call Parent class constructor by specifying the error name.
* Using pre-defined functionality, pre defined exception class creates object.

/\*class RuntimeException

{

public RuntimeException(String message)

{

Constructs a new runtime exception with the specified detail message.

}

}\*/

class CustomException extends RuntimeException

{

CustomException(String name)

{

super(name);

}

}

class CreateException

{

public static void main(String args[])

{

new CustomException("Error-Message");

}

}

**Throw:**

* It is a keyword.
* It is used to throw an exception object explicitly by the programmer.
* Pre-defined exceptions will be raised(thrown) by JVM automatically.
* Custom exception must be raised manually by the programmer.
* If we don’t handle the exception, object will be submit to “Default Exception Handler” program.

class CustomException extends RuntimeException

{

CustomException(String name)

{

super(name);

}

}

class ThrowException

{

public static void main(String args[])

{

CustomException obj = new CustomException("Error-Message");

throw obj ;

}

}

* Above custom exception is Runtime Exception
* Compiler is not giving error in case of not handling.
* Unchecked exception handling is optional.

**Custom checked Exception:**

* Extending from Exception class.
* Handling is mandatory.
* Compiler raises error if we don’t handle exception.

class CustomException extends Exception

{

CustomException(String name)

{

super(name);

}

}

class ThrowException

{

public static void main(String args[])

{

CustomException obj = new CustomException("Error-Message");

throw obj ;

}

}

**Handling Custom Exception:**

* Generally exceptions will raise inside the method while executing the logic.
* Whenever we are calling the function, we need to handle the exception.

Throws:

* It is a keyword.
* It is used to represents, a method raising exception.
* It is giving information that, we need to handle exception while calling the function.

class CustomException extends Exception

{

CustomException(String name)

{

super(name);

}

}

class Test

{

static void func() throws CustomException

{

CustomException err = new CustomException("Error-Name");

throw err;

}

}

class ThrowException

{

public static void main(String args[])

{

try

{

Test.func(); // calling

}

catch (CustomException e)

{

System.out.println("Exception : " + e.getMessage());

}

}

}

**General Example – Exception in Account transaction:**

import java.util.Scanner;

class LowBalanceException extends Exception

{

LowBalanceException(String name)

{

super(name);

}

}

class Account

{

private int balance;

Account(int amount)

{

this.balance = amount;

}

public int getBalance()

{

return this.balance ;

}

void withdraw(int amount) throws LowBalanceException

{

System.out.println("Trying to withdraw : " + amount);

System.out.println("Balance is account : " + this.balance);

if(amount <= this.balance)

{

System.out.println("Collect cash : " + amount);

this.balance = this.balance - amount;

}

else

{

throw new LowBalanceException("Low balance");

}

}

}

class Bank

{

public static void main(String[] args)

{

int amount;

Scanner scan = new Scanner(System.in);

System.out.print("Enter initial amount : ");

amount = scan.nextInt();

Account acc = new Account(amount);

System.out.println("Balance is : " + acc.getBalance());

System.out.print("Enter withdraw amount : ");

amount = scan.nextInt();

try

{

acc.withdraw(amount);

}

catch (LowBalanceException e)

{

System.out.println("Exception : " + e.getMessage());

}

System.out.println("Final balance : " + acc.getBalance());

}

}

**Assignment:**

1. Read 2 integer values from command line and collect

(**ArrayIndexOutOfBoundsException** if they are not giving input)

1. Convert input(strings) values into integers.

(**NumberFormatException** in case of invalid input)

1. Perform division operation and print.

(**ArithmeticExcepiton** in case the denominator is zero)

**Solution:**

class Division

{

public static void main(String args[])

{

try

{

String s1 = args[0];

String s2 = args[1];

int x = Integer.parseInt(s1);

int y = Integer.parseInt(s2);

int z = x/y;

System.out.println("Result : " + z);

}

catch(ArrayIndexOutOfBoundsException e1)

{

System.out.println("Exception : Insufficient input values");

}

catch(NumberFormatException e2)

{

System.out.println("Exception : Only integers allowed as input");

}

catch(ArithmeticException e3)

{

System.out.println("Exception : Division by zero");

}

}

}

**Output:**

D:\>javac Division.java

D:\>java Division

Exception : Insufficient input values

D:\>java Division abc xyz

Exception : Only integers allowed as input

D:\>java Division 10 0

Exception : Division by zero

D:\>java Division 10 5

Result : 2